

MOTOR GRÁFICO CON PORTALES PARA SIMULAR ESCENAS 3D NO EUCLIDIANAS

...

GRAPHICS ENGINE WITH PORTALS TO
SIMULATE NON-EUCLIDEAN 3D SCENES

— Diego Mateos Arlanzón —

GRADO EN DESARROLLO DE VIDEOJUEGOS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID



Trabajo Fin de Grado en Desarrollo de Videojuegos

Curso académico 2019-2020

Septiembre de 2020

Directora del trabajo:

Ana Gil Luezas

Autorización de difusión

Diego Mateos Arlanzón

18 de Septiembre de 2020

El abajo firmante, matriculado en el **Grado en Desarrollo de Videojuegos** de la **Facultad de Informática**, autoriza a la **Universidad Complutense de Madrid (UCM)** a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores el presente *Trabajo Fin de Grado*: “**MOTOR GRÁFICO CON PORTALES PARA SIMULAR ESCENAS 3D NO EUCLIDIANAS**”, realizado durante el curso académico **2019-2020** bajo la dirección de **Ana Gil Luezas** en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Agradecimientos

A Ana, por su paciencia, ayuda y confianza a la hora de dirigir este TFG.

Al resto de profesores y compañeros de la facultad, por acompañarme durante este grado.

A mi familia, por su apoyo y ánimo incesable.

— Resumen —

Este proyecto se centra en la simulación de escenas 3D en las que pueden existir *portales*. Un portal es un fenómeno físico que afecta al espacio de la escena, de tal forma que las propiedades necesarias para poder seguir siendo considerada euclidiana dejan de cumplirse. Los portales se manifiestan por pares y esencialmente conectan dos superficies separadas de dicho espacio; esto consecuentemente permite ver y desplazarse a través de ellos, independientemente de la distancia.

El objetivo principal de este proyecto es generar un recurso público completo, riguroso y autocontenido, sobre el funcionamiento y la implementación de estos portales en un motor gráfico sencillo. Para poder desarrollar el proyecto, ha sido necesario realizar una amplia investigación sobre casos existentes de implementación (principalmente sobre la saga de videojuegos *Portal*), y las tecnologías disponibles para elaborar mi propia versión.

En esta memoria se explica de forma accesible y didáctica todo lo necesario para comprender el fenómeno de los portales. Incluso se introducen los fundamentos de la informática gráfica y concretamente de *OpenGL*, para que una persona ajena al tema, si lo desea, pueda seguir también la parte técnica de la explicación. Respecto a mi implementación propia, se aporta íntegramente el código fuente, y la exposición concreta sobre los portales se mantiene suficientemente abstracta para ser reproducible con facilidad en la arquitectura de cualquier otro motor.

Palabras clave

Motor gráfico, Portales, Renderizado, Escena 3D, Informática gráfica, OpenGL, Shader.

— Abstract —

This project focuses on the simulation of 3D scenes in which *portals* may exist. A portal is a physical phenomenon that affects the space of the scene, in such a way that the properties necessary to continue to be considered Euclidean are no longer fulfilled. Portals manifest in pairs and essentially connect two separate surfaces from said space; this consequently allows you to see and move through them, regardless of distance.

The main objective of this project is to generate a complete, rigorous and self-contained public resource about the inner workings and implementation of these portals in a simple graphic engine. In order to develop the project, it has been necessary to carry out extensive research on existing implementation cases (mainly on the *Portal* videogame saga), and the technologies available to develop my own version.

In this dissertation, everything necessary to understand the phenomenon of the portals is explained in an accessible and didactic way. Even the fundamentals of computer graphics and specifically *OpenGL* are introduced, so that a person outside the subject, if they wish, can also follow the technical part of the explanation. Regarding my own implementation, it is fully detailed, but the specific exposition about the portals remains sufficiently abstract to be easily reproducible in the architecture of any other engine.

Keywords

Graphics engine, Portals, Rendering, 3D Scene, Computer graphics, OpenGL, Shader.

Índice general

1. Introducción	1
1.1. Motivación	1
1.2. Objetivos y plan de trabajo	3
1.3. Estructura de la memoria	3
2. Estado del arte: <i>Informática gráfica</i>	4
2.1. Definición de motor gráfico	4
2.2. Librerías de bajo nivel para gráficos 3D	6
2.3. Comparativa de librerías: <i>OpenGL</i>	8
2.4. Fuentes de información	10
2.5. Conclusión	11
3. Arquitectura de la aplicación: <i>Motor</i>	12
3.1. Tecnologías utilizadas	12
3.2. Estructura lógica	13
3.3. Conclusión	15
4. Fundamentos de informática gráfica	16
4.1. Tubería de renderizado	16
4.2. Máquina de estados	18
4.3. <i>Buffers</i> de memoria	20
4.4. Transformaciones y sistemas de coordenadas	23
4.5. Conclusión	29
5. Estado del arte: <i>Portales</i>	30
5.1. Definición de portal	30
5.2. Caso de estudio	33
5.3. Fuentes de información	34
5.4. Métodos de implementación de portales	36
5.5. Comparativa de métodos	39
5.6. Conclusión	43
6. Arquitectura de la aplicación: <i>Portales</i>	44
6.1. Vista a través de la superficie	46
6.2. Desplazamiento en primera persona	53
6.3. Vista externa de un desplazamiento	58
6.4. Vista recursiva a través de portales	66
6.5. Conclusión	76

7. Conclusiones y trabajo futuro	77
7.1. Conclusiones	77
7.2. Trabajo futuro	78
 Bibliografía	 79
 Apéndice A. Introduction [EN]	 84
A.1. Motivation	84
A.2. Goals and work plan	85
A.3. Structure of the memory	85
 Apéndice B. Conclusions and future work [EN]	 86
B.1. Conclusions	86
B.2. Future work	87
 Apéndice C. Controles de la aplicación [ES]	 88

1. Introducción

En este breve capítulo se presentan las motivaciones detrás del proyecto, sus objetivos concretos y el plan de trabajo seguido. También se explica esquemáticamente la estructura completa de la memoria.

El código fuente asociado a este proyecto se encuentra en el siguiente repositorio (que también ha sido añadido a las referencias [1]): https://github.com/dimateos/TFG_Portals.

1.1. Motivación



[Figura 1.1.1]: *Fotograma del corto de animación “Portal: No escape” [2]. Se muestra una estancia en la que se han dispuestos dos portales, uno enfrente del otro, creando una recursión infinita. Ese efecto se consigue con efectos especiales añadidos en post-producción.*

Los portales son un fenómeno interesante del que se pueden encontrar ejemplos en diversas producciones de todo tipo de industrias: animación, cine, videojuegos, etc. Entre estos destacan los portales utilizados en videojuegos, ya que normalmente sus propiedades se definen rigurosamente y se simulan en tiempo real. Pero por desgracia, su implementación técnica es compleja y la información disponible acerca de ella es limitada y dispersa (y totalmente inexistente en español). De este hecho surge el deseo de realizar una extensa investigación para poder llevar a cabo el desarrollo de mi propia implementación. Además, la intención en este proyecto es partir desde cero y crear un motor gráfico sencillo; esto permite ampliar mis conocimientos sobre la informática gráfica y motores en general.



[Figura 1.1.2]: *Captura de pantalla del videojuego “Portal” [3]. De nuevo, se muestra una estancia en la que se han dispuestos dos portales, uno enfrente del otro, creando una recursión infinita. En este caso el efecto se simula a tiempo real y es interactivo.*

El propósito final del proyecto es solventar la falta de información concreta encontrada inicialmente. Esto se pretende conseguir mediante la redacción de un documento que explique el funcionamiento y la implementación de los portales en un motor gráfico sencillo. Así, esta memoria en sí misma, junto con el asociado código fuente de mi implementación, compondrían un recurso público que destaca por ser completo, riguroso y autocontenido. Para que la explicación sea lo más accesible y didáctica posible, incluso se introducen los fundamentos de la informática gráfica y la tecnologías utilizadas para que una persona ajena al tema, si lo desea, pueda seguir también la parte técnica de la exposición. Respecto a mi implementación propia, se detalla íntegramente, para que pueda ser consultada de forma útil. Y finalmente la explicación concreta referente a los portales se mantiene suficientemente abstracta para ser reproducible con facilidad en la arquitectura de cualquier otro motor.

1.2. Objetivos y plan de trabajo

Los objetivos de este proyecto se pueden listar ordenados por prioridad. Se tiene en cuenta la posibilidad de no poder completarlos todos debido a una carga de trabajo superior a la anticipada. Este mismo orden se ha seguido como plan de trabajo:

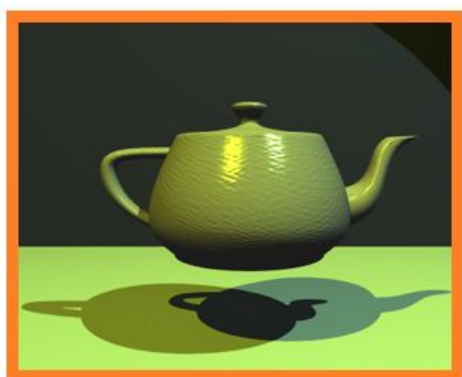
1. Estudiar cómo funcionan los portales y métodos de implementación existentes.
2. Estudiar los fundamentos de la informática gráfica requeridos para implementarlos.
3. Desarrollar un motor gráfico sencillo reutilizable con capacidades lógicas suficientes.
4. Implementar una escena 3D interactiva como base para el desarrollo.
5. Implementar los portales de forma progresiva, empezando por sus propiedades más básicas, hasta las más complejas.
6. Redactar la parte de la memoria referente a los fundamentos de la informática gráfica, para que sea más accesible, y para aliviar la carga de explicación posterior.
7. Redactar la parte de la memoria referente al completo funcionamiento e implementación de los portales, lo más didáctica y útil posible.
8. Redactar la parte de la memoria referente a la arquitectura del motor propio, para que sea posible consultar el código fuente de forma productiva.
9. Implementar mejoras sobre el motor gráfico, principalmente soporte para iluminación y para importación de modelos o escenas completas.
10. Desarrollar escenas más impresionantes, que empleen los portales de formas sorprendentes e interesantes.

1.3. Estructura de la memoria

Este documento trata de ser muy accesible y por ello parte del mismo puede ser información redundante para una persona que ya domine el tema. Concretamente los capítulos [2] y [4] referentes a la informática gráfica, se pueden leer por encima si no se requiere una explicación sobre su estado del arte y fundamentos. El capítulo [3] detalla brevemente la arquitectura del motor propio, si ésta no es de interés, puede ser saltado y consultado en caso de duda. Finalmente los relativamente extensos capítulos [5] y [6] explican el estado del arte de los portales y su funcionamiento e implementación completamente.

2. Estado del arte: *Informática gráfica*

La informática gráfica es una rama de las ciencias de la computación que se especializa en procesar o modificar imágenes capturadas del mundo físico y también en crear imágenes generadas por ordenador. Es una amplia área que ha desarrollado una gran cantidad de software y hardware especializado. Esta tecnología ha tenido un impacto significativo y revolucionado todo tipo de industrias y medios de comunicación; actualmente resulta indispensable para animación, fotografía digital, diseño gráfico, cine, videojuegos, etc. Otro ejemplo concreto que ilustra cómo forma parte de la vida cotidiana, es el hardware específico que controla y potencia la mayoría de pantallas de toda clase de dispositivos.



[Figura 2.0]: *Imagen de la “Tetera de Utah” por Martin Newell [4]. Creada en 1975, esta tetera y sus imágenes renderizadas, son emblemáticas para el desarrollo de gráficos por ordenador. Usar un modelo de esta tetera es considerado como un equivalente a escribir un programa “Hello World!”.*

La informática gráfica depende profundamente, además de en la tecnología de la computación en general, de teorías y metodologías existentes en otras ciencias como la geometría, óptica y física. En esta memoria, se comienza con la definición de conceptos generales relevantes, conectados con los objetivos del proyecto. Posteriormente se estudian varias tecnologías actuales especializadas que permiten desarrollarlo y sus diferencias.

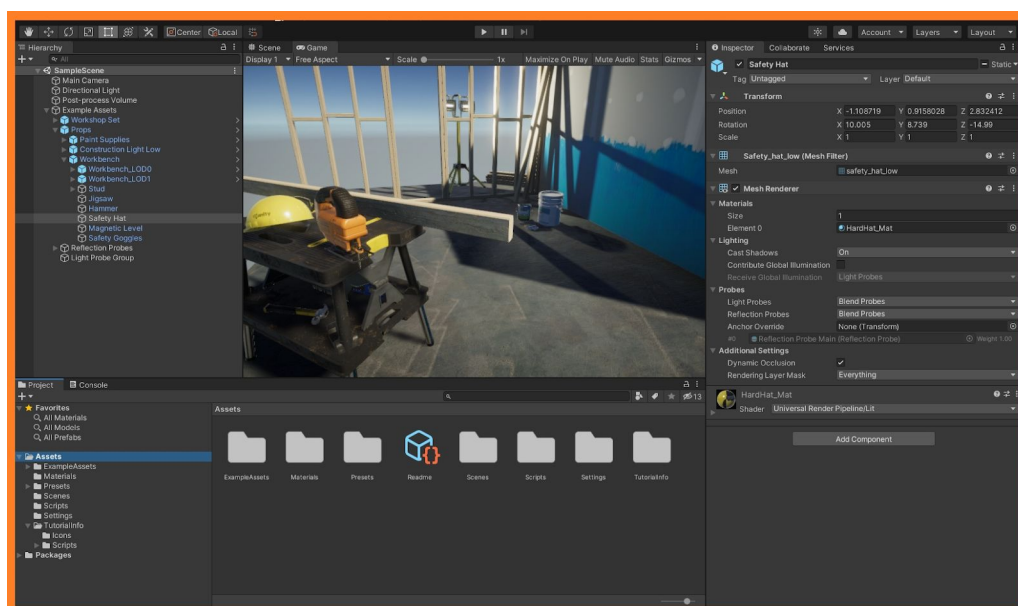
2.1. Definición de motor gráfico

Los videojuegos hoy en día no se realizan desde cero, suelen empezar con un *motor* ya existente. Un motor de videojuego es un software que recoge y junta toda la tecnología base necesaria para crear un juego: motor gráfico para renderizar gráficos 2D y 3D, motor físico, motor de sonido, animación, soporte para lenguajes de *scripting*, etc. Normalmente también cuentan con interfaces visuales para ayudar con el diseño y programación del juego.

Hace años, los motores eran privados y cada empresa solía desarrollar uno propio para sus productos. Actualmente existen motores públicos comerciales como *Unity* [5] o *Unreal Engine* [6] de los que dependen incluso grandes desarrolladoras de videojuegos. Entre las principales ventajas que aportan se encuentran las siguientes:

1. Un juego realizado con estos motores puede ser exportado con mayor facilidad a múltiples plataformas. Equipos de desarrollo están detrás de ellos y los mantienen al día para las plataformas principales de la industria.
2. Cuentan con una gran comunidad que genera una amplia cantidad de recursos para aprender a utilizarlos, resolver dudas y sacar su máximo potencial.

Pero estos grandes motores son de carácter general, y no son adecuados para todo tipo de proyectos, ya que a veces no se necesita toda la tecnología que aportan e incluso pueden llegar a ser contraproducentes. En estas situaciones, es mejor crear uno propio sencillo, con las características requeridas concretas y las tecnologías preferidas. Además, estos desarrollos suelen suponer una gran experiencia didáctica, ya que se adquiere un mayor control sobre todo el motor, incluido a bajo nivel. Estos beneficios se alinean con los objetivos de este proyecto, ya que se centra especialmente en una parte específica del motor de juego, el motor gráfico o motor de renderizado. Esta parte es la encargada de dibujar en pantalla todas las imágenes necesarias para representar el estado del juego. El programador modifica las propiedades de todos los elementos que componen la escena que se simula y se abstrae de las rutinas necesarias para renderizarlos. Los motores gráficos se desarrollan utilizando librerías de bajo nivel para gráficos 3D; estas librerías son el tema del que trata el siguiente apartado.



[Figura 2.1]: *Captura de pantalla de la interfaz gráfica del editor del motor de videojuegos Unity [5] (obtenida de su web). La jerarquía de elementos de la escena se muestra arriba a la izquierda y sus propiedades se editan en el menú de la derecha. En el centro se puede observar la vista renderizada de todos los objetos. Algunos motores también permiten al usuario o al programador escoger entre más de una librería gráfica disponible.*

2.2. Librerías de bajo nivel para gráficos 3D

Los gráficos 3D, con el paso de los años, han llegado a ser tan utilizados (particularmente en videojuegos) que ha dado lugar a la aparición de librerías especializadas. Estas librerías proporcionan una interfaz a los programadores, que facilita todas las etapas del proceso de generación de imágenes por ordenador y el uso de aceleración utilizando hardware específico. Las interfaces han demostrado ser vitales para los fabricantes, ya que permiten acceder a cualquier característica de una tarjeta gráfica de forma abstracta, independientemente del fabricante del hardware o modelo de GPU.



[Figura 2.2.1]: *Foto de un producto comercial actual, Gigabyte GeForce RTX 2060 [7]. Un modelo de tarjeta gráfica lanzada en 2018 por Nvidia [8], manufacturado por la empresa Gigabyte [9]. Las tarjetas gráficas se han convertido en una pieza del hardware de un ordenador muy importante. En el mercado compiten una gran cantidad de modelos y fabricantes distintos.*

A estas interfaces de programación, también se les conoce por las siglas API (en inglés, “*application programming interface*”). Existen varias librerías de bajo nivel para gráficos 3D, pero sólo unas pocas destacan ya que son las más utilizadas y apoyadas por empresas de la industria. Algunas llevan muchos años en desarrollo y han pasado por diferentes etapas y versiones. Esta evolución ha sido motivada paralelamente por diversos cambios y mejoras en el diseño del hardware, adaptándose a las crecientes necesidades del consumidor final. A continuación, se describe brevemente tres de las API más populares actuales:

- **OpenGL** [10]: su primera versión fue publicada en 1992, desarrollada por *Silicon Graphics*. Desde 2006 sigue en desarrollo por *Khronos Group* [11], un grupo sin ánimo de lucro. Su última versión estable es la “4.6” de 2017. Es una especificación de software implementada por los fabricantes de tarjetas. Es multiplataforma y tiene interfaces para muchos lenguajes de programación [12], incluido C++ [13]. En el pasado, la mayoría de tarjetas gráficas sólo daban soporte a *OpenGL* y por ello fue el estándar. Es utilizada tanto en videojuegos como en otros ámbitos profesionales de la informática gráfica (animación, modelado 3D, visualización científica, simulación, etc).
- **Vulkan** [14]: su primera versión fue publicada en 2016 y la última estable es la “1.2.146” de 2020. Es un software de licencia libre desarrollado por *Khronos Group* [11]. Es multiplataforma y tiene interfaces para varios lenguajes de programación (no tantos como *OpenGL*), incluido C++. *Khronos Group* lo presentó como el “*OpenGL* de la siguiente generación”. Entre las diferencias más destacables se encuentran [15]: acceso a más bajo nivel (con código más verboso), misma API para plataformas móviles y de escritorio, menor uso de CPU y la capacidad de paralelización utilizando varios núcleos del procesador. Actualmente, *Vulkan* es utilizado principalmente solo en videojuegos.
- **Direct3D** [16]: su versión inicial fue publicada en 1996, desarrollada por *Microsoft*. Su última iteración es *Direct3D 12* versión “2004” de 2020. Es un software propietario de *Microsoft*. Es para el sistema operativo *Windows* y tiene interfaces para pocos lenguajes de programación (esencialmente los creados por *Microsoft*), incluido C++. La iteración más reciente fue desarrollada con el objetivo de lograr las mismas ventajas que ofrece *Vulkan*, las versiones anteriores competían con *OpenGL* por el mercado. En el caso de *Direct3D*, también es utilizado principalmente solo en videojuegos.

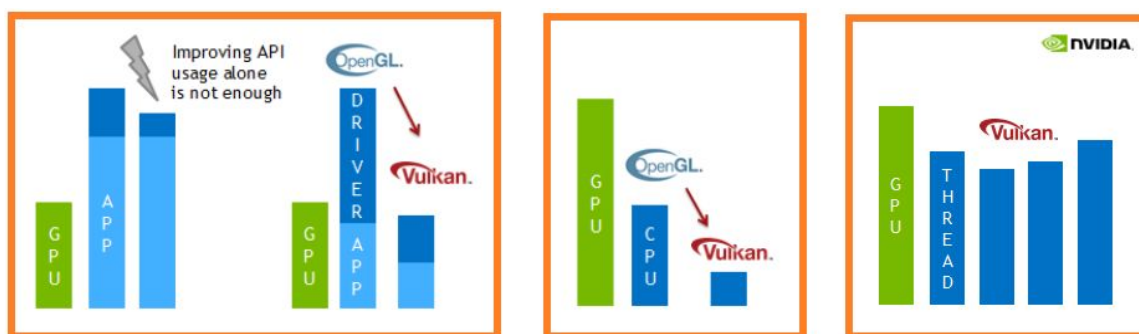


[Figura 2.2.2]: Logos actuales de las API mencionadas. *Direct3D* se distribuye como parte de *DirectX* (conjunto de interfaces para manejo de entrada, ventanas, etc).

2.3. Comparativa de librerías: *OpenGL*

El objetivo de este proyecto es investigar y exponer el proceso de desarrollo de portales de forma abstracta y reproducible para cualquier API gráfica, pese a ello, es necesario escoger una de las disponibles. Con la API seleccionada, se implementa un sencillo motor gráfico propio en el que realizar experimentos y prototipos. En este proyecto se ha decidido usar *OpenGL* para dicha implementación.

Desde el punto de vista del rendimiento, como hemos visto previamente, *Vulkan* y *Direct3D* ofrecen una mayor capacidad ya que aportan medidas para disminuir el trabajo de la CPU (acceso a más bajo nivel, paralelización, etc). Estas ventajas llevan consigo el inconveniente de requerir un código bastante más complejo. *OpenGL* necesita menos líneas para pintar en pantalla y además alivia el trabajo al programador realizando ciertas tareas implícitamente (comprobación de errores, compilación de programas, evitar que se borren recursos que están siendo usados, etc) [17]. Si la aplicación no está limitada por el uso de CPU, la utilización de *Vulkan* o *Direct3D* no aporta un impacto al rendimiento suficiente como para justificar el mantenimiento de su código (por ello se usan principalmente sólo en videojuegos). Utilizando *OpenGL*, obtenemos una velocidad de desarrollo superior que complementa el objetivo del proyecto de investigar y prototipar. Se puede plantear la realización de una implementación en otra API gráfica como trabajo futuro una vez se hayan cumplido los objetivos del proyecto.



[Figura 2.3]: Gráficas representativas obtenidas de Nvidia [8]. En la izquierda, se observa el ejemplo de una aplicación con una gran carga de trabajo en la CPU (en parte debida a OpenGL), que se reduce al utilizar Vulkan. En el centro, se aprecia cómo Vulkan por sí mismo, no reduce apenas la carga de trabajo de la GPU. Finalmente en la derecha, se muestra que con Vulkan es posible distribuir la carga de trabajo de CPU entre varios hilos.

Desde el punto de vista de los recursos y fuentes de información, *OpenGL* es la más antigua y la que cuenta una mayor cantidad disponible con diferencia (en el apartado siguiente veremos una lista). Es común que un desarrollador opte por aprender las bases de la informática gráfica estudiando *OpenGL*. A esto hay que añadir el hecho de que yo ya parto con conocimientos previos sobre esta librería, pues durante el año académico 2017-2018 cursé la asignatura Informática Gráfica I (IG1) [18]. En ésta se introduce la API y distintos conceptos del campo de la informática gráfica con prácticas realizadas utilizando *OpenGL*.

Se ha de destacar que *OpenGL* lleva muchos años en desarrollo y existen múltiples versiones que añaden grandes cambios a la librería [19]. El objetivo de IG1 era dar una introducción, por ello se usó una versión antigua en la que las etapas del proceso de dibujado son fijas, y con ello menos flexibles, pero es más sencillo programar y continuar aprendiendo. La intención en este proyecto es usar una versión moderna de *OpenGL* con etapas programables (igual que en las otras librerías populares), esto implica que es necesario realizar cierta investigación y aprendizaje previo. Las diferencias se explican en detalle en el posterior capítulo [4] “*Fundamentos de la informática gráfica*”.

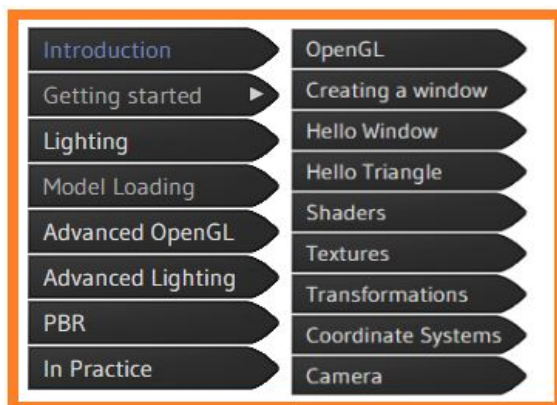
Parte de la investigación se agiliza pues en 2018-2019 cursé a la asignatura Informática Gráfica II (IG2) [20]. En esta continuación de IG1, se explican conceptos más avanzados utilizando el motor gráfico de código abierto *OGRE 3D* [21], incluidos los *shaders* [22] (“*sombreadores*” en español, pero comúnmente se utiliza la palabra inglesa). Los *shaders* se escriben externamente a la aplicación y son los programas que se inyectan en las mencionadas etapas programables de *OpenGL*. Por ello, ya estoy familiarizado con los tipos de *shaders* y el lenguaje *GLSL* (*OpenGL Shading Language* [23] en el que se pueden escribir.

Para desarrollar una aplicación capaz de dibujar utilizando una API gráfica, además de elegir una, también se debe decidir que lenguaje, herramientas, librerías complementarias y estructuras se van a utilizar. Estas se tratan brevemente en el siguiente capítulo [3] “*Arquitectura de la aplicación: Motor*”.

2.4. Fuentes de información

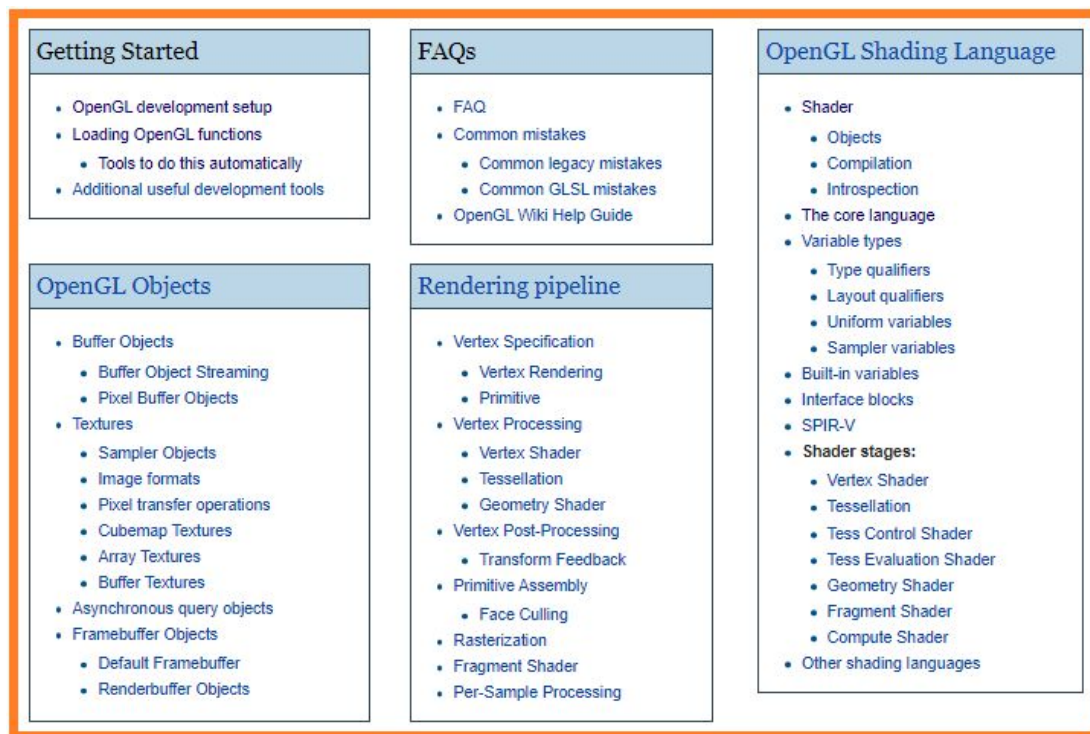
En internet se encuentran muchos recursos y tutoriales de libre acceso disponibles para sobre *OpenGL*. En este caso se han priorizado los recursos más completos y en formato escrito.

- El recurso más utilizado ha sido “*Learn OpenGL*” [24], que cuenta con página web y un libro [25] escrito por su autor, *Joey de Vries*. Ésta ha sido la principal fuente de información para el aprendizaje e investigación sobre *OpenGL* moderno. Posteriormente, durante el desarrollo del proyecto, también se ha necesitado para consultar conceptos y detalles concretos. El contenido es de gran calidad, explica paso a paso *OpenGL*. Todos los ejemplos de código están disponibles en su repositorio [26] de *Github* bajo la licencia *CC BY-NC 4.0* [27] y las imágenes bajo *CC BY 4.0* [28].



[Figura 2.4.1]: *Pequeña muestra del índice de contenido de “Learn OpenGL” [24]. Se destaca la pestaña de “Getting started” por la que se comienza el aprendizaje. Múltiples conceptos del apartado “Advanced OpenGL” son necesarios para desarrollar los portales.*

- “*Modern OpenGL Guide*” [29] es otro recurso de calidad que también cuenta con página web y versión en e-book, por su autor *Alexander Overvoorde*. Los ejemplos de código y el libro están disponibles en su repositorio [30] de *Github* bajo la licencia *CC BY-SA 4.0* [31]. Esta fuente de información se ha utilizado para contrastar datos y aclarar dudas.
- La *OpenGL Wiki* [32] reúne una gran colección de información sobre *OpenGL* y su API. También incluye tutoriales y preguntas comunes. Es el recurso accedido para resolver cuestiones técnicas. Otras fuentes de información técnica relacionadas con las tecnologías utilizadas se exponen brevemente en un apartado posterior de la memoria. La más consultada de éstas es la documentación de la API [33] de *GLM*.



[Figura 2.4.2]: Esquema del contenido de la “OpenGL Wiki” [32]. Entre las distintas secciones, se encuentran las mencionadas etapas de dibujado (“Rendering pipeline”) y el lenguaje GLSL (“OpenGL Shading Language”). Varios de los objetos de la sección “OpenGL objects” se explican en apartados posteriores de la memoria.

- Para conceptos más avanzados o muy concretos, se ha buscado y accedido también a otros recursos puntualmente, con intención de contrastar y ampliar información. En estos casos, las fuentes importantes se referencian en los apartados relacionados.

2.5. Conclusión

En este capítulo se ha descrito lo que es un motor de videojuegos, un motor gráfico y una librería de bajo nivel para gráficos 3D (*API*). Tras conocer estos conceptos, se han presentado las principales API actuales, y entre ellas se escoge *OpenGL* de forma razonada. Finalmente, se listan las fuentes de información más relevantes que se han utilizado durante el estudio e investigación relativo a la informática gráfica. A partir de este punto, ya se cuenta con un conocimiento base sobre la tecnología fundamental de este proyecto

3. Arquitectura de la aplicación: *Motor*

Para desarrollar cualquier proyecto de software se deben plantear las tecnologías y estructuras lógicas que se van a utilizar. Este trabajo final de grado requiere la implementación de un motor gráfico sencillo para posteriormente poder investigar y prototipar el proceso de desarrollo de portales.

Como ya se ha especificado previamente, el objetivo final no es explicar dicho desarrollo concreto, sino elaborar una “guía” suficientemente abstracta que recoja y explique todo el conjunto de conceptos y procedimientos necesarios para implementar portales. Por ello, en este capítulo no se entra en demasiado detalle pues se sale del ámbito de la memoria. El código fuente asociado se encuentra en el repositorio del proyecto [1].

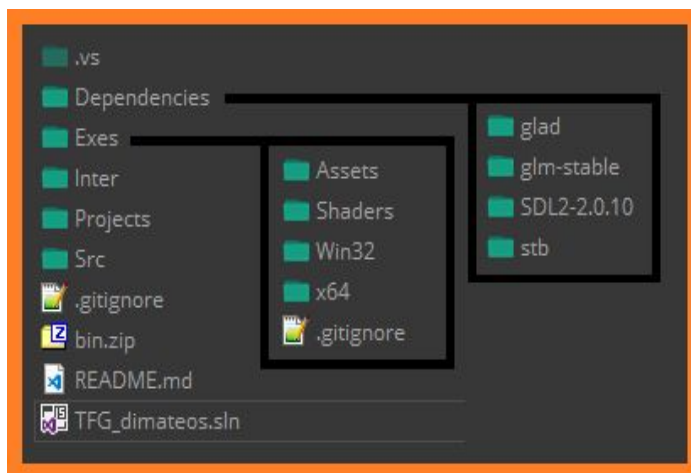
3.1. Tecnologías utilizadas

Se utiliza *OpenGL* [10] con el lenguaje de programación con el que cuento con más experiencia, *C++* [13] (ya usado previamente con *OpenGL*). A continuación se lista las diferentes librerías utilizadas, todas ellas multiplataforma y con licencias permisivas e incluso de código abierto:

- **SDL 2.0** [34]: la librería *SDL* (*Simple DirectMedia Layer*) se utiliza para la creación de ventanas y acceso a eventos de entrada de usuario. En la anteriormente mencionada como principal fuente de información para aprender *OpenGL* (*LearnOpenGL*) [24], se usa la librería *GLFW* [35], pero ya que yo ya he trabajado con *SDL* en múltiples ocasiones y me parece más completo he preferido incorporarlo a este motor.
- **glad** [36]: *glad* es una librería de carga de *OpenGL* (también puede cargar otras librerías gráficas). Existen otras librerías de carga disponibles [37]. De forma muy resumida, estas librerías son necesarias para cargar los punteros a las funciones de *OpenGL* al principio de la ejecución del programa. Estas funciones se deben cargar en ejecución ya que a diferencia de como ocurre normalmente, no son estáticas, dependen del hardware.

- **GLM** [38]: *OpenGL Mathematics* es una librería matemática para software de computación gráfica. Proporciona clases y funciones para realizar transformaciones de matrices, cuaterniones, etc. Se usa como base para la mayoría de operaciones.
- **stb_image** [39]: *stb_image* forma parte de un conjunto de librerías de un único archivo para C/C++. *Stb_image* permite cargar imágenes en multitud de formatos de forma sencilla pero flexible. Se emplea para cargar las texturas de la aplicación.

La aplicación como tal se ha desarrollado en *Visual Studio* [40], *IDE* (Entorno de Desarrollo Integrado) comercial propiedad de *Microsoft* con el que ya he completado múltiples proyectos en el pasado. Con este software se compila el código y genera el ejecutable final enlazando todas las librerías necesarias. Para terminar, queda añadir que se ha utilizado el sistema de control de versiones *Git* [41] a través de la plataforma *Github* [42].



[Figura 3.1]: Imagen del directorio que contiene la solución del proyecto (archivo “.sln”) y una cuidada estructura de subdirectorios para el resto de archivos. Se muestran las dependencias agrupadas por carpetas (tecnologías mencionadas). En el directorio de ejecutables se separan los distintos recursos y binarios de cada plataforma.

3.2. Estructura lógica

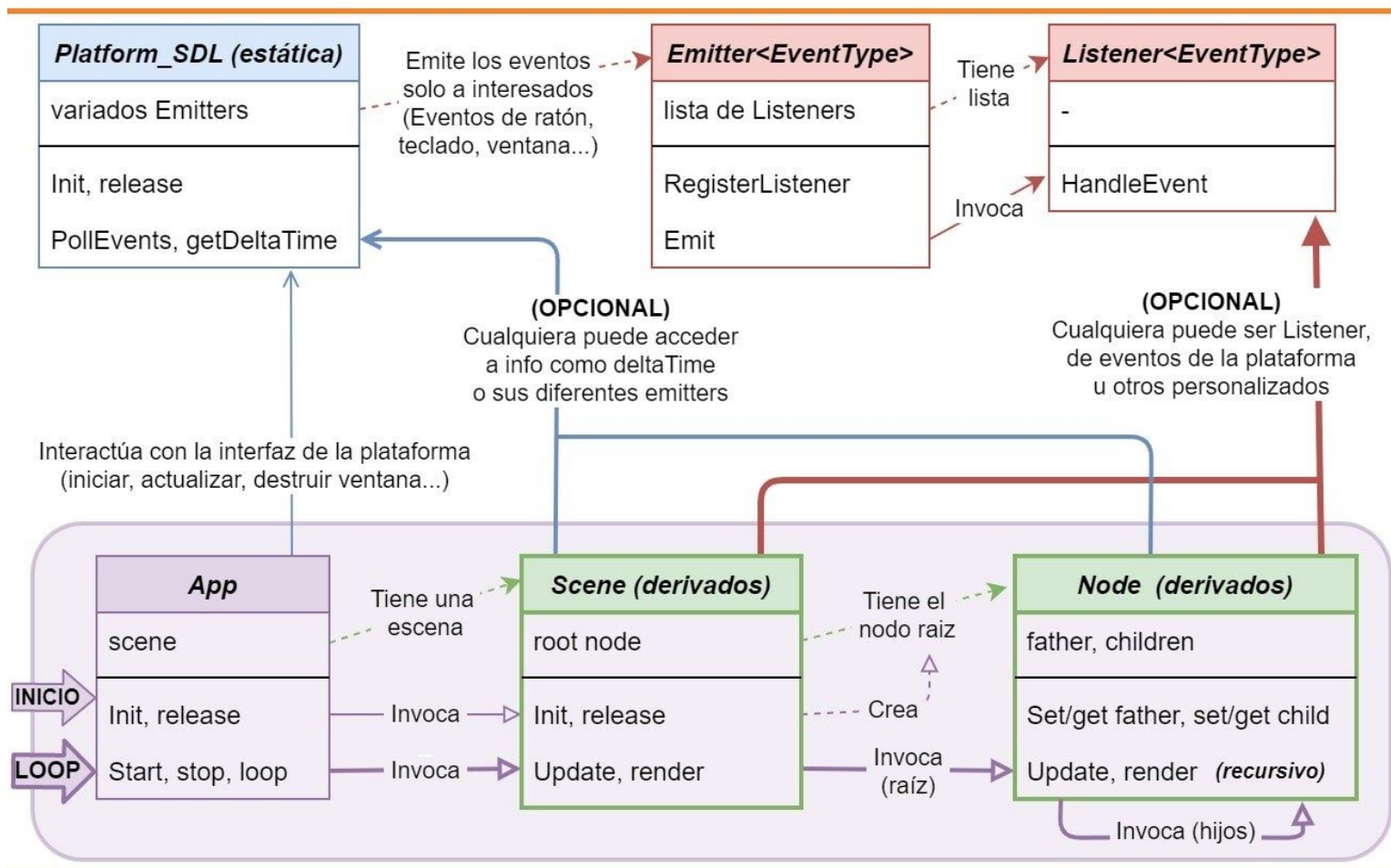
Como todos los programas que funcionan en tiempo real, esta aplicación emplea un bucle principal que ejecuta constantemente la lógica requerida en cada ciclo hasta su detención final. Esto se conoce comúnmente como el patrón de diseño de programación “*Update Method*” [43] o “*Game Loop*” [44]. En este caso se utiliza orientación a objetos y métodos separados *Update* y *Render* para, respectivamente, actualizar el estado y dibujarlo ciclo a ciclo. La unidad básica es la clase *Node* y sus derivados implementan variaciones específicas de *Update* y/o *Render*. A su vez todos los objetos *Node* forman parte de un objeto

clase *Scene*, este controla parte del flujo de ejecución y maneja todo tipo de interacciones. Finalmente la propia *Scene*, es creada desde la clase *App*, que es el punto de entrada de la aplicación. La relación entre estas clases se muestra esquematizada en la figura [3.2].

Para interactuar con la plataforma se utiliza una clase pública estática llamada *Platform_SDL*, que ofrece la interfaz necesaria e interiormente está implementada haciendo uso de la librería *SDL*. Esta clase también da acceso global a variables como *deltaTime* [45] (tiempo transcurrido desde el ciclo anterior del bucle principal). Existen otras dos clases importantes para la estructura lógica, *Emitter* y *Listener*, que permiten transmitir eventos (del tipo configurado) a otros objetos interesados en recibirlos. Este es el conocido patrón de diseño “Event-Subscriber” [46]. En este caso *Platform_SDL* posee varios objetos *Emitter*, que utiliza para enviar de forma separada los diferentes tipos de eventos que produce la plataforma (pulsaciones de teclado, ratón, ventana...), sólo a aquellos registrados para recibirlos.

Esta estructura es más compleja de lo normal ya que lo usual es enviar los eventos a todos los objetos (*Node* o *Scene*) y que estos los consuman o no. Cuando un objeto lo haya consumido se detiene la propagación, pero en el peor de los casos un evento que no es de interés pasa por todos y cada uno de ellos sin consumirse. En cambio, en este motor en cada ciclo del bucle principal, *App* indica a *Platform_SDL* que proceda a enviar todos los eventos producidos y este los *emite* de forma separada y sólo si hay *listeners* registrados. Por ejemplo, si ningún objeto está interesado en el movimiento del ratón, toda esa gran cantidad de eventos se descarta anticipadamente. Otro detalle poco común pero sencillo es la omisión de *deltaTime* como parámetro de los métodos de actualización, su valor se obtiene directamente de *Platform_SDL* por los objetos interesados. Estas mejoras mencionadas no aportan demasiado beneficio en rendimiento, pero han sido realizadas como ejercicio de aprendizaje.

La organización de objetos *Node* de la *Scene* también es interesante ya que es un *árbol* [47] (*estructura de datos jerárquica*) en vez de la usual simple lista. *Scene* tan sólo tiene referencia al *Node raíz* y recorre el árbol en *profundidad-primero* (*pre-orden*) [48] para llamar a sus funciones *Update* y *Render* (dos veces, primero se actualizan todos y después se dibujan). Esta forma de representar una escena es bastante común ya que la relación jerárquica que se establece ayuda a gestionarla. En este proyecto, esta complejidad añadida está justificada.



[Figura 3.2]: Esquema simplificado de la estructura lógica de la aplicación. En la parte inferior se encuentran las clases que forman parte directa del loop del programa, y en la superior la interfaz con la plataforma y los transmisores de eventos.

3.3. Conclusión

Tras finalizar este capítulo se tiene una idea suficientemente cercana de la arquitectura y entorno de desarrollo empleado en este proyecto, así como de la investigación y planeamiento que ha requerido. A partir de este punto, es posible consultar código fuente del repositorio [1] de forma bastante más productiva. En el [Apéndice C] se puede encontrar más información relacionada con los controles y funcionalidades de la aplicación desarrollada.

4. Fundamentos de informática gráfica

Antes de comenzar a definir que es un portal y cómo implementarlo, se debe conocer ciertos conceptos de la informática gráfica. No se trata de entrar en detalles técnicos, sino de explicar su aplicación y funcionamiento de forma general. La finalidad de este capítulo es comprender los mecanismos e ideas en los que se apoya el desarrollo de los portales para, posteriormente, lograr un mejor entendimiento. Para profundizar en estos fundamentos se puede consultar los siguientes libros [49] [50].

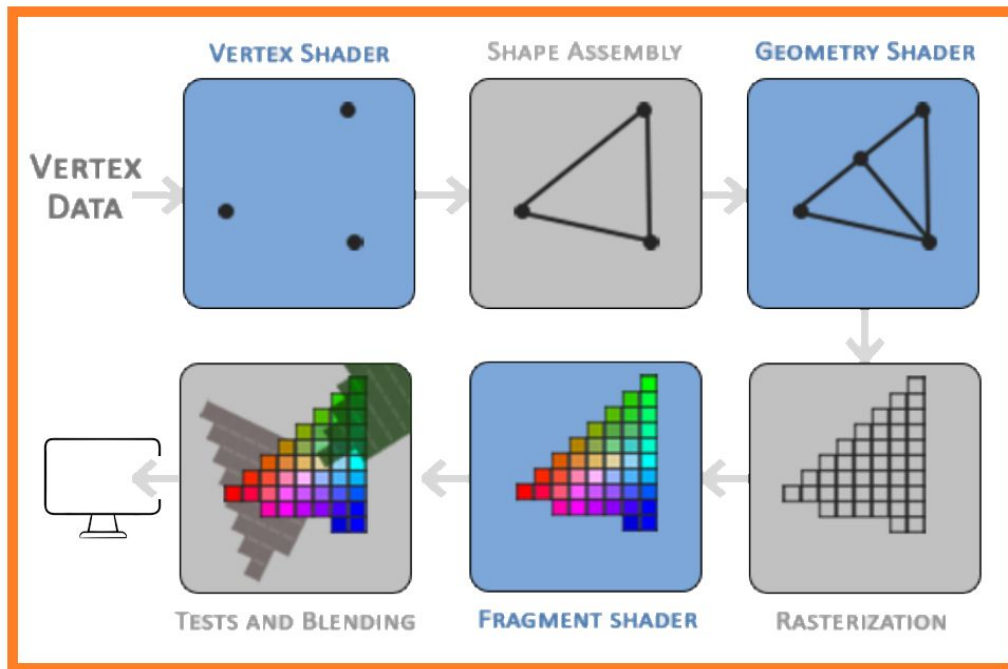
Se toma como base *OpenGL*, pero los mismos conceptos están presentes en la mayoría de API gráficas, aunque varíe su nombre. Lo común es que difieran en su uso desde el código. Por ejemplo, existen múltiples formas de crear la ventana del programa en la que dibuja *OpenGL*, en el motor de este proyecto se utiliza la librería mencionada *SDL*.

4.1. Tubería de renderizado

En *OpenGL* todo se representa con vértices en un espacio 3D, pero la ventana de la aplicación está formada por una matriz de píxeles 2D. La tubería de renderizado (en inglés, “*graphics pipeline*”) de *OpenGL*, es la que se encarga de transformar las coordenadas 3D en los píxeles 2D adecuados. Es un proceso largo y se divide en una serie de etapas concretas, donde cada una de ellas necesita como entrada de datos la salida de la etapa anterior.

Cada etapa tiene una única función específica y por ello es fácil lograr su ejecución en paralelo. Debido a esta potencial paralelización, las GPU modernas contienen una gran cantidad de pequeñas unidades de procesamiento especializadas en procesar los datos de dichas etapas. Cada una de las etapas la realiza un pequeño programa distinto ejecutado en la GPU. Todos estos programas son lo que se denomina *shader*.

A continuación se muestra una representación de las etapas de la tubería de renderizado y posteriormente se explica cada una de forma simplificada para poder entender mejor el funcionamiento general del proceso. Cada paso realiza una parte específica de la transformación de vértices a píxeles.



[Figura 4.1]: Recurso editado de “Learn OpenGL” [24]. Representación abstracta de las etapas de la tubería de renderizado.

- **Vertex Shader:** recibe como entrada los datos de un único vértice (normalmente posición, color y coordenadas de textura). Los vértices 3D reciben diferentes transformaciones para pasar a *NDC* (del inglés, “*normalized device coordinates*”), un espacio de coordenadas 3D cuyos valores varían en el intervalo $[-1.0, 1.0]$. Este espacio ayuda a simplificar proyecciones y el recorte de primitivas fuera de dichos valores.
- **Shape Assembly:** recibe todos los vértices individuales que definen una primitiva básica y los ensambla, es decir los agrupa para la siguiente etapa.
- **Geometry Shader:** recibe la colección de vértices de la etapa anterior y es capaz de generar otras primitivas (por ejemplo, más triángulos), creando nuevos vértices.
- **Rasterization:** recibe las primitivas y calcula los píxeles individuales correspondientes en pantalla, aquellos que se ajustan al interior de su forma. Con cada uno de estos, se forma un *fragmento* (en inglés, “*fragment*”), que reúne toda la información que necesita *OpenGL* para dibujar un pixel final (coordenadas, color, profundidad, etc).
- **Fragment Shader:** recibe cada fragmento y calcula el color definitivo del píxel. Este se modifica utilizando texturas o información de la escena como luces, sombras, etc.
- **Test and Blending:** finalmente, para cada fragmento se realizan ciertas comprobaciones opcionales, activadas o no por el desarrollador. Las más comunes consisten en determinar si el fragmento se encuentra detrás de algún objeto opaco (debe ser descartado) o translúcido (se debe de mezclar sus colores acordemente).

En versiones antiguas de *OpenGL* la tubería de renderizado era fija, es decir que el desarrollador no podía editar las funciones que se realizaba en cada etapa. Esta diferencia ya es mencionada previamente en la memoria. Hoy en día es posible escribir programas que reemplacen a los *shaders* por defecto, y con ello lograr que el color final de un pixel no tenga nada que ver. Las etapas configurables se destacan en la figura [4.1] con un fondo azul, en realidad es *obligatorio* definir mínimo un *Vertex Shader* y un *Fragment Shader*; este último es el *shader* más utilizado para crear todo tipo de efectos. Utilizar el *Geometry Shader* es opcional y menos frecuente, y en versiones avanzadas de la API existe el *Tessellation Shader* [51]. El resto de API destacadas también permiten el uso de *shaders* personalizados.

4.2. Máquina de estados

OpenGL funciona como una gran máquina de estados, una larga colección de variables indican cómo debe operar. Existen funciones que varían su estado y funciones que ordenan realizar operaciones con dicho estado. Por ejemplo, antes de dibujar un polígono se cambia la anchura de las líneas a dibujar, dicha variación no sólo afecta a ese polígono, sino a todas las operaciones de dibujado que utilicen líneas hasta que se vuelva a determinar de nuevo otra anchura. Explicado con una metáfora: cuando se cambia el color y la forma del pincel (se configura el estado), se usa siempre el mismo para realizar todos los dibujos (se opera) hasta que se indique lo contrario. Tener esto en mente ayuda a entender mejor cómo funciona *OpenGL* y el propósito detrás de la mayoría de sus estructuras y objetos.

Las tarjetas gráficas modernas son muy rápidas y por ello es interesante tratar de reducir la interacción entre CPU y GPU, es decir enviar una menor cantidad de comandos siempre que sea posible. Antiguamente existía un cuello de botella en las API gráficas ya que en algunos casos se tardaba más en transmitir comandos de configuración de estado que en realizar las operaciones de pintado. Explicado con la metáfora previa: se pasaba más tiempo cambiando el pincel (configurando el estado), que utilizándolo para dibujar (completando las operaciones). Ésta es una de las grandes diferencias con las nuevas generaciones de librerías de bajo nivel para gráficos 3D. En particular, *Khronos Group*, que lleva el desarrollo y mantenimiento de *OpenGL*, lanzó en 2016 *Vulkan* con la idea de crear el “*OpenGL* de la siguiente generación”. Este comienza con otra base de código, es más complejo y entre otras cosas no tiene máquina de estados global, pero consigue un rendimiento gráfico superior.

En las últimas versiones, *OpenGL* es más flexible y no existe tanto cuello de botella. Como se razona previamente en la memoria, aunque no se obtenga siempre el mismo rendimiento que con *Vulkan* o *Direct3D 12*, para determinadas aplicaciones sencillas no hay diferencia sustancial. Para configurar la máquina de estados global de forma más eficiente, siempre que se puede se agrupa la información para transmitirla toda junta a la tarjeta gráfica. Si es posible, esta información o parte de ella se almacena en la memoria de la propia GPU, para posteriormente poder mandar un volumen menor de datos que tiene la opción de referenciar información ya contenida en la GPU, si lo necesita.

En general, los objetos de una escena se representan con un conjunto de listas de vectores. Un cubo puede ser una lista de coordenadas (vector 3D) para la posición de cada vértice (respecto al origen del objeto), una lista de colores (vector RGB) para el color de cada vértice y una lista de coordenadas de textura (vector 2D) para asignar un determinado color a un vértice dada una imagen. En realidad, todos estos valores individuales se empaquetan en una única lista de elementos y se indica a *OpenGL* como interpretarla. Es muy configurable, se puede utilizar distintas precisiones, cambiar el orden e incluso añadir o quitar propiedades, por ejemplo es común que si se usan coordenadas de textura no se indique el color de los vértices.



[Figura 4.2]: Recurso editado de “*Learn OpenGL*” [24]. Representación abstracta de una lista de valores que representa las propiedades de un objeto (posición y color).

Los *shaders* también se empaquetan. Los personalizados, escritos por el desarrollador, se leen de su fichero de texto, se compilan y finalmente se juntan un único programa que representa toda la tubería de renderizado configurada. Este programa empaquetado también se suele denominar *shader* (pero agrupa todos), y representa una importante variable de la máquina de estados. Por ello, lo ideal es usar pocos *shaders* diferentes y agrupar los objetos que compartan el mismo, para pintarlos seguidos y así minimizar los cambios de *shader* por ciclo.

Los *shaders* son altamente configurables, pero tiene que cumplir ciertos requisitos. Ya que es la etapa inicial, la entrada de datos del *Vertex Shader*, debe coincidir exactamente con el formato de datos que representa cada objeto (posición, color, coordenadas, etc). Y como se puede suponer, también es vital que los datos de salida de este *shader* coincidan con la entrada del *Fragment Shader*. Además de entradas directas, existe la posibilidad de añadir otras variables al programa para configurar su estado, denominadas *uniform*, que se pueden editar exteriormente y mantienen un valor constante para todos los vértices procesados.

4.3. *Buffers de memoria*

En informática, un *buffer* (“*búfer*” en español, pero comúnmente se utiliza la palabra inglesa) es un espacio de memoria en el que se almacenan datos de manera temporal, normalmente se usa para evitar que se pierda acceso a cierto recurso durante una transferencia. Otro caso de uso, como los que veremos a continuación, es para evitar cuellos de botella en procesos muy rápidos.

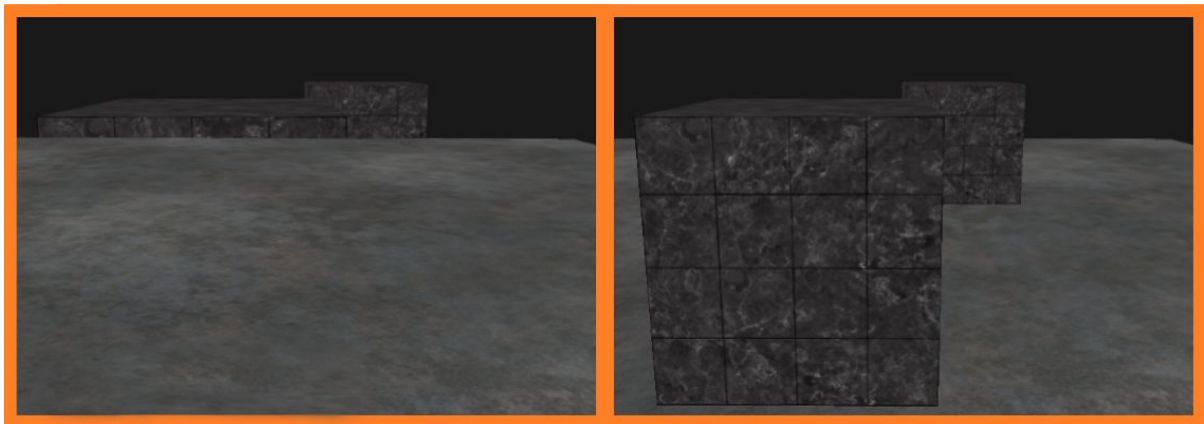
Las llamadas a la API de *OpenGL* para dibujar no escriben directamente el color de cada píxel que sale de la tubería de renderizado en la pantalla, sino que estas llamadas realizan sus operaciones de escritura sobre un *buffer*. Normalmente se trabaja con más de un *buffer* para no tener que esperar a que se pinte la pantalla completa, ya que si sólo se cuenta con uno y no se espera, se puede producir el artefacto visual “*tearing*”. Teniendo dos, por ejemplo, uno puede representar la matriz de la pantalla (conocido como *front-buffer*) mientras se escribe en el otro (*back-buffer*). Una vez procesados todos los objetos de la escena, se intercambian los *buffers* y se empieza a escribir en el que antes representaba la pantalla (ahora *back-buffer*). Su contenido ya no importa, ya que en ese momento puede estar desactualizado respecto a la situación de la escena.



[Figura 4.3.1]: Captura de pantalla del videojuego Portal [3]. Ejemplo de “*tearing*”. Este artefacto ocurre cuando la actualización del estado de la escena afecta al búfer que está siendo dibujado. También puede ocurrir a nivel de hardware sino se sincroniza con la tasa de refresco del monitor.

Cuando se intercambian *buffers* y el contenido se descarta, lo normal es borrarlo completamente antes de empezar a dibujar de nuevo la escena. Para borrar se escribe el mismo valor de color en todo el *buffer*. En *OpenGL* existen otros tipos de *buffer* que no son para color, estos sirven para realizar operaciones opcionales que descarten o combinen píxeles. Estas comprobaciones tienen lugar en la última etapa de la tubería de renderizado (ya mencionadas, en “*Test and Blending*”). Estos *buffers* también se borran antes de dibujar y tienen el mismo tamaño que los *buffers* que representan la pantalla, esto permite asociar cada uno de sus valores a un píxel concreto.

El más utilizado es el *depth-buffer*, que permite gestionar la profundidad de los fragmentos y discernir cuales se encuentran ocluidos. Cada vez que se va a escribir un nuevo fragmento, se compara su valor de profundidad con el ya escrito en el *depth-buffer*, si es menor lo sobrescribe y además escribe el color del fragmento en el *back-buffer*. Durante esta operación, también se tiene en cuenta el *alpha* de los fragmentos para mezclar los colores si es necesario. Cabe destacar aunque sea un poco abstracto, que los valores de profundidad de cada fragmento no corresponden linealmente a su distancia en la escena. El objetivo de esto es tener una enorme precisión en los objetos cercanos, y consecuentemente, una minúscula en los lejanos.



[Figura 4.3.2]: Recurso compilado de “*Learn OpenGL*” [24]. Comparación entre la misma escena con *depth-buffer* desactivado (izda.) y activado (dcha.). Siendo la derecha la vista habitual, en la vista izquierda, el suelo se ha dibujado por encima del resto de elementos. Esto se debe a que el suelo ha sido el último objeto en ser dibujado y no se ha descartado ningún fragmento a pesar de tener los cubos delante.

Aunque su uso sea mucho menos frecuente, el *stencil-buffer*, es útil para lograr ciertos efectos. Al igual que el *depth-buffer* tiene la opción de descartar fragmentos antes de escribirlos en el *back-buffer*. En su caso, es más configurable y se comprende mejor con un caso de uso, por ejemplo el reflejo de una escena sobre una superficie reflectante:

1. Si se utiliza *depth-buffer*, se puede empezar con el dibujo de la escena normal.
2. Antes de dibujar la superficie reflectante, se activa la escritura en el *stencil-buffer* y se desactiva la escritura en el *depth-buffer*.
3. Se dibuja dicha superficie horizontal y todos sus fragmentos dejan una marca en el *stencil-buffer*, actualizando su contenido y sin tocar el *depth-buffer*.
4. Posteriormente se desactiva la escritura en el *stencil-buffer* y se activa que a partir de ese momento se descarten los fragmentos no marcados.
5. Finalmente se procede a dibujar todos los elementos de la escena boca-abajo (invertida su escala Y y desplazados si es necesario).



[Figura 4.3.3]: Recurso compilado de “Modern OpenGL Guide” [29]. A la izquierda se muestra un intento fallido de reflejo, el cual es ocluido por el suelo, además de aparecer fuera de su superficie. A la derecha se observa el resultado de seguir los pasos indicados para obtener un reflejo sencillo utilizando el *stencil-buffer*.

Tanto el *depth-buffer* como el *stencil-buffer*, son opcionales. Si el desarrollador los utiliza debe indicar cuándo se borran y activan o desactivan, para crear los efectos deseados. También puede escoger su precisión, cómo y cuándo un fragmento se descarta o se escribe y modifica sus valores, etc. La combinación de todos estos *buffers*, incluido el de color, se guarda en una region de la memoria de la GPU y se denomina *framebuffer*.

Por defecto sólo se tiene un *framebuffer* que permite dibujar en la ventana, pero *OpenGL* da la flexibilidad de crear otros *framebuffers*, con diferentes tamaños o *depth-buffer* asignados, etc. A primera vista puede aparentar no tener mucho sentido, pero realizar las operaciones de escritura de dibujado de la escena, sobre un *framebuffer* personalizado, es la base de muchos efectos. Aunque no se dibuje nada directamente en la pantalla, la representación correcta de la escena ya se encuentra almacenada en una zona de memoria, y ésta se puede utilizar como textura. Una vez obtenida la textura que representa la escena, se le puede aplicar infinidad de efectos de postprocesado (escala de grises, color invertido, etc), antes de pintarla finalmente en el *framebuffer* por defecto de la pantalla. Otro uso particular interesante, es utilizar dicha textura en la superficie de un objeto dentro de la misma escena, para lograr algún tipo de reflejo o ventana.

4.4. Transformaciones y sistemas de coordenadas

Como se ha explicado, el *Vertex Shader* recibe la lista de datos que representa un objeto y como parte de ella, se encuentran las coordenadas 3D de sus vértices. Para desplazar un objeto, no se editan esos vértices (es costoso), sino la posición de su centro de coordenadas y al *shader* se añade una variable *uniform* para mandarle dicha posición. Teniendo ambos vectores en el *shader*, se puede obtener la posición final del vértice realizando su suma.

Un objeto no sólo se traslada, sino que también se escala y rota, por ello no se usa un simple vector para representar sus transformaciones, se usan matrices. Las propiedades y operaciones de vectores y matrices son extensas, por ello en esta memoria sólo se centra en su aplicación y no en los razonamientos matemáticos. Un vector se puede representar como una matriz $[N \times 1]$ donde $[N]$ es el número de dimensiones del vector. El elemento neutro del producto de matrices se llama matriz identidad, $[I_n]$ donde $[n]$ es el tamaño de la matriz.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot 2 \\ 1 \cdot 3 \\ 1 \cdot 4 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

[Figura 4.4.1]: “*Learn OpenGL*” [24]. La multiplicación entre una matriz identidad y cualquier vector resulta en el vector inalterado. Se puede decir que el vector es *pre-multiplicado* por la matriz.

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

La traslación y escala de un objeto se puede representar con una matriz de forma sencilla.

[Figura 4.4.2]: “Learn OpenGL” [24]. Encima, un vector se pre-multiplica por una matriz de translación (vector T_x , T_y , T_z). Debajo, otro vector se pre-multiplica por una matriz de escala (vector S_1 , S_2 , S_3).

La rotación también se puede representar con matrices, pero es bastante más complicado, sobre todo de visualizar. Además, para poder rotar alrededor de un eje arbitrario, se requiere combinar rotaciones en cada eje y esto introduce el problema de *gimbal-lock* [52]. No se discuten los detalles, pero una mejor solución es utilizar *cuaterniones* [53], que son mejores computacionalmente y evitan este problema.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos \theta \cdot y - \sin \theta \cdot z \\ \sin \theta \cdot y + \cos \theta \cdot z \\ 1 \end{pmatrix}$$

[Figura 4.4.3]: “Learn OpenGL” [24]. Ejemplo de rotación alrededor del eje X utilizando una matriz.

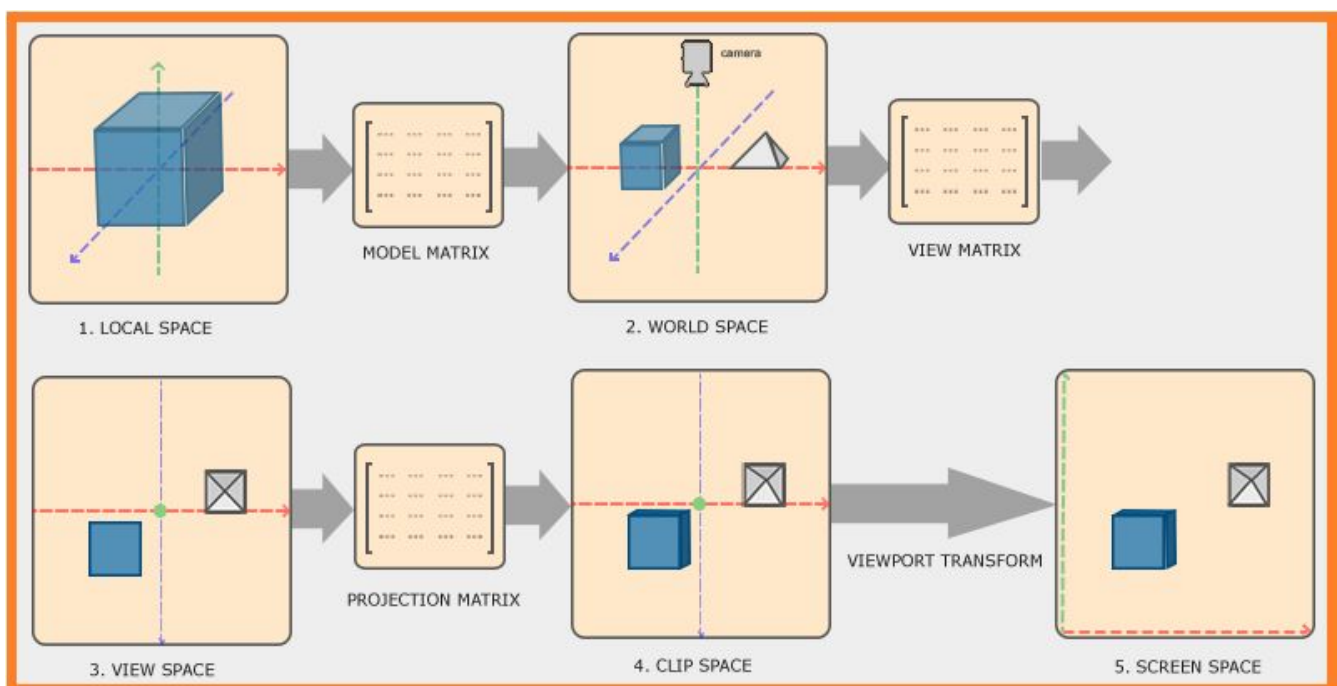
En la práctica, todas estas matrices de transformación se combinan en una única utilizando las propiedades del producto de matrices. El orden en dicho producto es muy importante ya que no es conmutativo. En esta situación, las matrices se ordenan de derecha a izquierda para establecer el orden de aplicación al objeto. El orden más habitual es escala $[S]$, rotación $[R]$ y traslación $[T]$, dando lugar a $[T \cdot R \cdot S]$. La matriz resultante *pre-multiplicada* a otra, afecta igual que cada una de ellas por separado en el orden establecido $[S \rightarrow R \rightarrow T]$. Si la rotación es computada en cuaterniones, se representa en forma de matriz previamente para poder realizar esta combinación.

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 2 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} 2x + 1 \\ 2y + 2 \\ 2z + 3 \\ 1 \end{pmatrix}$$

[Figura 4.4.4]: “Learn OpenGL” [24]. Ejemplo de combinación de matrices de traslación y escala. Ordenadas en orden inverso, la matriz resultante modifica el vector como si se aplicase primero la matriz de escala y después la de traslación.

Esta matriz final combinada, se denomina matriz de modelo (en inglés, “*model matrix*”). Se dice que representa la transformación del objeto (en inglés, “*transform*”), porque es equivalente a todas las transformaciones que necesita el objeto para situarlo en su estado final (partiendo desde su centro de coordenadas, con una escala y rotación neutra). Pero esta matriz de modelado, no es la única necesaria para determinar el estado en el que realmente se observa un objeto en pantalla, aún falta tener en cuenta la cámara, proyección, etc. Todos los vértices pasan por distintos sistemas de coordenadas, hasta por fin obtener sus coordenadas en el espacio de pantalla. Este proceso se realiza paso a paso con productos de matrices. A continuación se muestra un esquema global que posteriormente se explica.



[Figura 4.4.5]: Recurso editado de “Learn OpenGL” [24]. Esquema que muestra los sistemas de coordenadas por los que se pasa hasta llegar al espacio de pantalla.

1. **Espacio local:** Las coordenadas locales son relativas al origen del objeto. Los vértices que representan la forma de cada objeto están en este primer sistema de coordenadas.
2. **Espacio global:** Las coordenadas globales son relativas al origen de la escena. Para pasar un objeto de espacio local a global se *pre-multiplica* por su matriz de modelado.
 - Una escena repleta de objetos suele gestionarse con una jerarquía basada en un árbol, en la que poder establecer relaciones entre nodos padres e hijos. Entonces para todo nodo hijo, su matriz de modelado permite obtener sus coordenadas en el espacio local del padre.

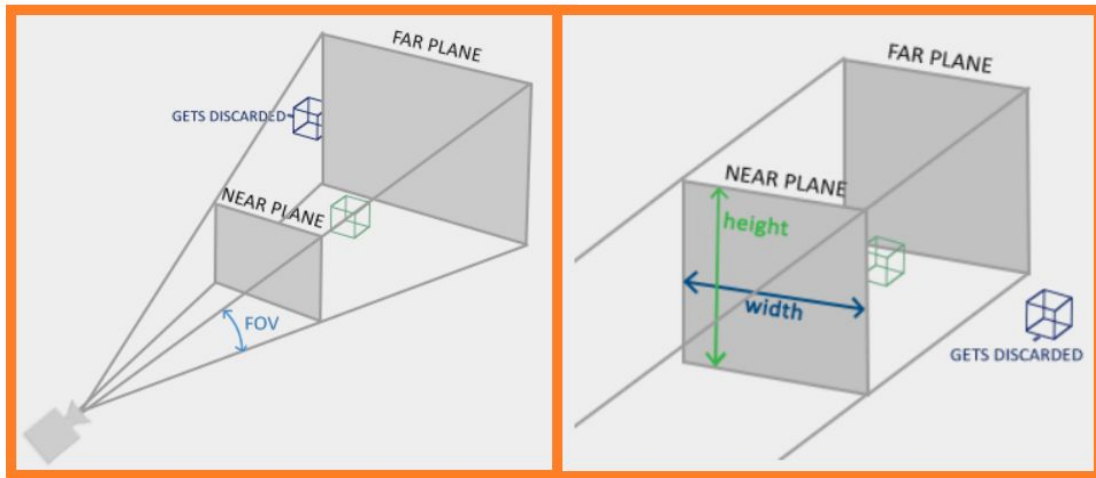
- Para obtener las coordenadas en espacio global, se debe *pre-multiplicar* por todas las matrices de modelado hasta el nodo raíz de la escena (sin padre).

Toda matriz de modelado de un objeto se puede transformar al sistema de coordenadas de otro. Esto se realiza con la matriz de **modelado inversa**, concretamente *pre-multiplicando* por ella. Esta matriz no es la inversa matemática, sino la obtenida al aplicar en orden contrario las inversas de su escala, traslación y rotación. Es decir que si la matriz de modelado es $[\mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}]$, la inversa es $[\mathbf{1}/\mathbf{S} \cdot \mathbf{R}^{-1} \cdot -\mathbf{T}]$. En este caso, como la rotación se representa con cuaterniones, se utiliza su expresión conjugada. La matriz de modelado inversa es un concepto complejo, pero lo importantes es entender para qué sirve: dados dos objetos con dos matrices de modelado representadas en el mismo sistema de coordenadas (escala, traslación y rotación desde el mismo origen), por ejemplo en el espacio global de la escena; es posible obtener la matriz de modelado de uno de ellos en el sistema de coordenadas local del otro (es otras palabras, teniendo la transformación actual de ese como su nuevo origen).

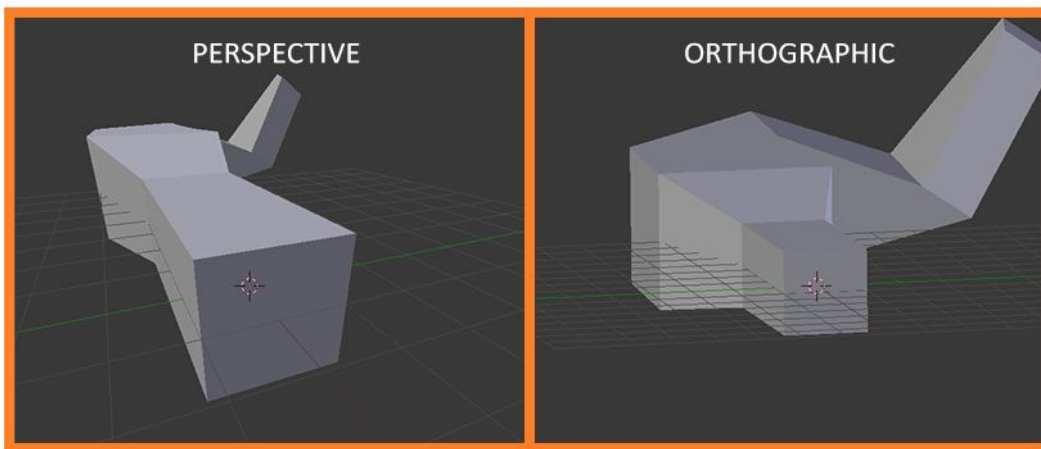
3. **Espacio de vista:** Las coordenadas en este espacio son relativas al punto de vista de la cámara. Para pasar de espacio global a vista, se pre-multiplica por la matriz de vista, que no es otra que la matriz de modelado inversa de la cámara.

- La transformación de la cámara se representa con una matriz de modelado como el resto de objetos. En este paso se transforman todas las escalas, traslaciones y rotaciones de los objetos para que estén en el sistema de coordenadas de la cámara.

Análogo a como en una cámara fotográfica es posible configurar múltiples parámetros, las cámaras en informática gráfica, también son configurables. Por ejemplo, el ángulo de visión^[1] en fotografía permite ajustar la magnitud de la parte capturada de una escena; en informática gráfica, es necesario definir exactamente el espacio capturado que se proyecta en pantalla. Por razones técnicas, lo más importante es definir un plano cercano y uno lejano que limiten la distancia mínima y máxima de la región visible; una menor separación de estos mejora la precisión de los cálculos de profundidad. También se necesita definir el contorno de esta región, que puede ser un prisma (definiendo anchura y altura) o un tronco (pirámide truncada definiendo un ángulo); la forma de este contorno afecta a la perspectiva final.



[Figura 4.4.6]: Recurso compilado de “Learn OpenGL” [24]. Representación de dos cámaras y sus planos que definen la región visible. Se comparan los dos contornos y la información necesaria para definirlos.



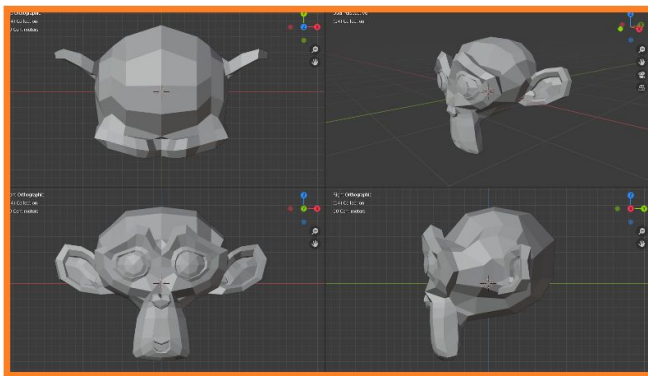
[Figura 4.4.7]: Recurso editado de “Learn OpenGL” [24]. Correspondiendo a las cámaras en la imagen anterior, se observa la perspectiva resultante. A la izquierda, una perspectiva natural con profundidad. A la derecha una perspectiva ortográfica^[1] sin profundidad.

4. **Espacio de clip:** Las coordenadas en este espacio se expresan en el rango especificado por los límites de la región del espacio visible de la cámara (definido por su configuración). Para pasar del espacio de vista a *clip* se debe pre-multiplicar por la matriz de proyección, que representa la configuración de la cámara. El nombre del espacio viene del verbo inglés “to clip”.
 - Cada vértice no contenido en el intervalo $[-1.0, 1.0]$, se descarta pues significa que no se encuentra dentro del volumen visible. En caso de ser parte de un objeto, *OpenGL* lo recorta y construye nuevos triángulos que encajan dentro del rango.

- Finalmente las coordenadas se encuentran en *NDC* (del inglés, “*normalized device coordinates*”), este es el rango en el que se encuentran los vértices al salir del *Vertex Shader*. Esta transformación y todas las anteriores tienen lugar en esta etapa de la tubería de renderizado.

5. **Espacio de pantalla:** El destino final del dibujado no necesariamente corresponde con el tamaño total de la ventana real, sino que se abstrae y se utiliza un *puerto de vista* (en inglés, “*viewport*”) que puede representar una porción de tamaño menor. Para pasar al espacio de pantalla se aplica la transformación que proviene de la configuración de dicho puerto de vista.

- La última operación que se realiza es la *perspective-division*, que coloquialmente se puede decir que aporta la perspectiva dividiendo las coordenadas por su distancia a la cámara: más lejos es más pequeño (en proyecciones ortográficas no tiene efecto).
- La coordenada *Z* se tiene en cuenta para los tests de profundidad y en la imagen final se representan píxeles con posiciones *X* e *Y* y color.



[Figura 4.4.8]: Sencilla muestra de una ventana con cuatro puertos de vista simultáneos. Su uso es común para mostrar varias perspectivas de un mismo objeto; por ejemplo, como en esta imagen de Blender [54], un software de modelado 3D.

Todas estas transiciones y sistemas de coordenadas diferentes proporcionan una amplia flexibilidad. Hay operaciones que son más sencillas de realizar en una capa que en otra. Por ejemplo, tiene sentido modificar un objeto en su espacio local pero realizar cálculos respecto a otros en espacio global. El efecto de trasladar la cámara es curioso, es el mismo que aplicar la misma traslación a todos los objetos de la escena en sentido contrario. Poder modificar la transformación y configuración de la cámara es innegablemente útil. Aun así, siempre es posible combinar varias matrices de transformación para no tener que aplicarlas por separado, pero esto conlleva la consecuente pérdida de flexibilidad.

4.5. Conclusión

Tener esta imagen general suficientemente detallada del proceso de dibujado y de determinados conceptos de la informática gráfica, es necesaria para entender los razonamientos y procedimientos que se exponen durante la explicación del desarrollo de portales. Es por ello que este capítulo alivia una enorme carga de información y permite centrarse mejor en los mecanismos específicos de los portales.

Desde el punto de vista de la implementación, se pueden añadir los siguientes apuntes:

- Todas las estructuras relacionadas con ángulos, vectores, cuaterniones, matrices y proyecciones se representan y operan utilizando la librería *GLM* [38].
- La cámara de la escena se puede controlar desde la lógica como en un FPS [55], esto quiere decir que se orienta moviendo el ratón y se desplaza con el teclado. Esto permite explorar la escena en primera persona. Tiene opciones para moverse respecto a la orientación de la cámara o respecto a la escena global. No hay gravedad ni físicas que limiten su movimiento, sólo se limita la rotación vertical.
- Nunca se utiliza el *framebuffer* por defecto, en cambio se usa uno intermedio para aplicar cualquier efecto de post-procesado.

Para terminar, un resumen del proceso de dibujado que tiene lugar en cada vuelta de bucle:

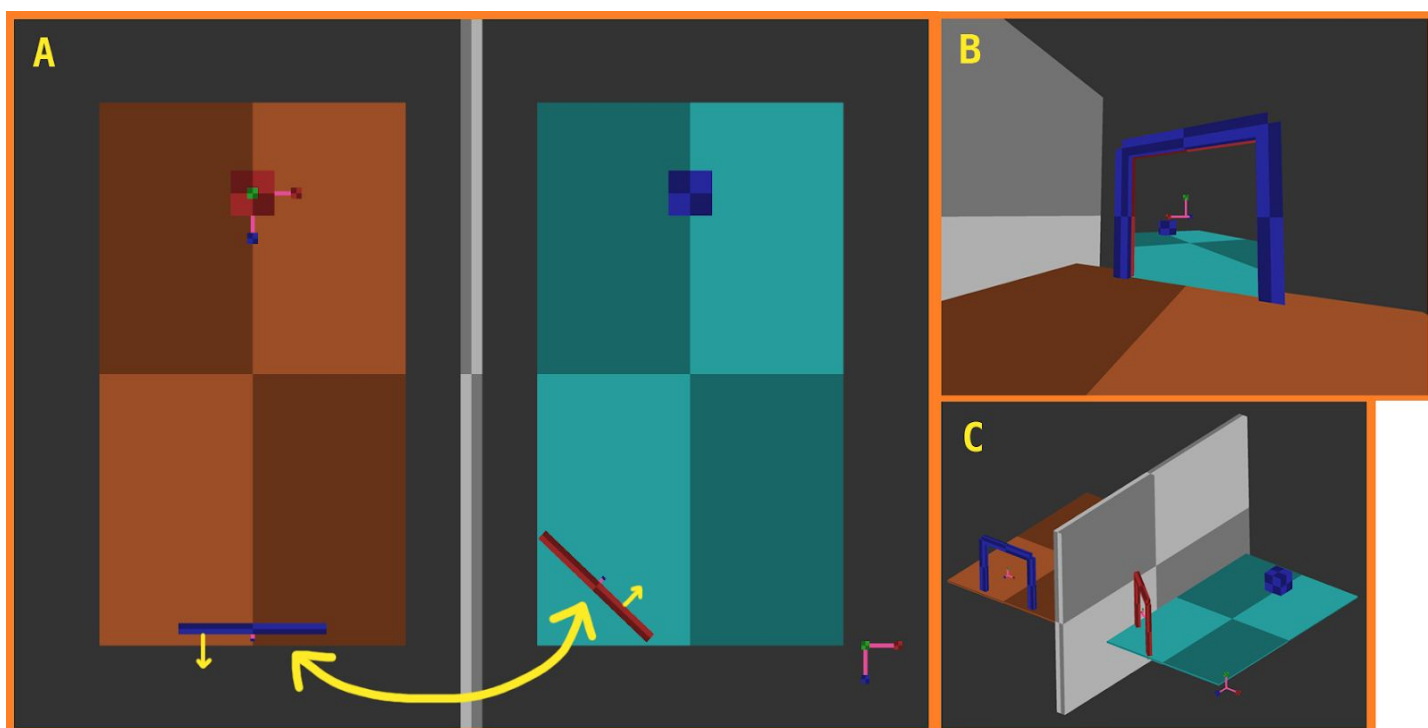
1. Se borran todos los *buffers* (color y profundidad, principalmente).
2. Se activa el *shader* deseado. Si se utilizan varios, los pasos siguientes son ejecutados de nuevo (pero sin repetir los objetos dibujados necesariamente).
3. Se asigna el valor a las variables *uniform* globales (comunes para todo objeto). Básicamente relacionadas con la cámara, como las matrices de vista y perspectiva.
4. Para cada objeto,
 - I. Se asigna el valor a las variables *uniform* necesarias, entre éstas se encuentra su matriz de modelado y textura (aunque la carga de ésta es un poco especial).
 - II. Se carga la información de sus vértices.
 - III. Se pide a *OpenGL* que dibuje.
5. Se procede a intercambiar los *buffers* (*back* por *front*) y actualizar la ventana.

5. Estado del arte: *Portales*

Este proyecto está enfocado en la simulación de una escena virtual con *portales*, realizada con las capacidades de la informática gráfica y la lógica necesaria. La noción coloquial de *portal* es bastante ambigua, por lo que antes de comenzar con su desarrollo, la primera necesidad es definirlo y especificar sus propiedades. Una vez definido, se puede proceder a presentar aquellas instancias en las que han sido previamente utilizados y se recopilan las fuentes de información disponibles más completas y convenientes.

5.1. Definición de portal

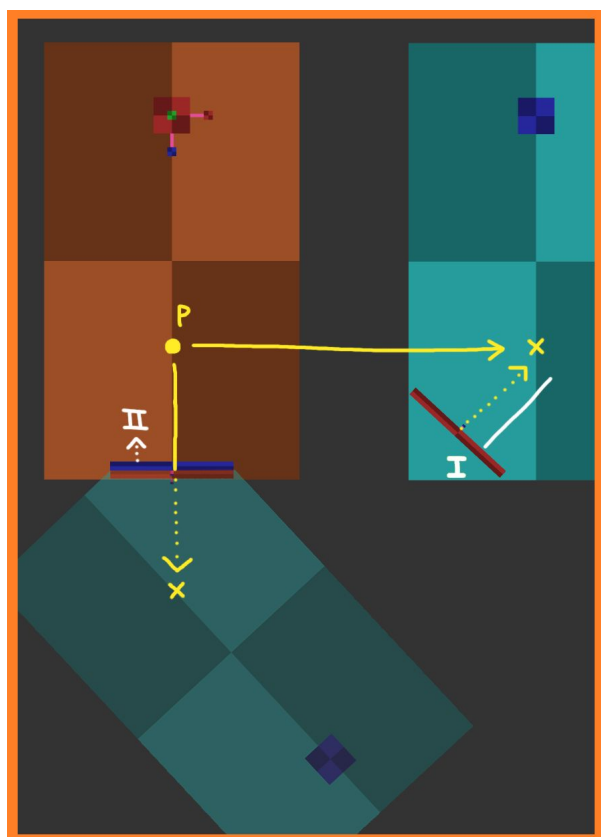
Un portal se refiere a una discontinuidad del espacio tridimensional euclidiano en la que una superficie delimita y conecta dos regiones de dicho espacio, que en realidad están separadas. Por simplificar, dicha superficie se mantiene plana y de forma rectangular; entonces se puede decir que su cara trasera se define como la cara trasera de otra superficie rectangular y viceversa, ambas localizadas en posiciones distintas del espacio. Cabe destacar que siempre hablaremos de pares de portales *conectados*. No es necesario que ni la luz, ni nada que atraviesa un portal se vea alterado; no es ningún vórtice temporal nebuloso extraño.



[Figura 5.1.1]: Esquema con varios puntos de vista de un ejemplo de portal en funcionamiento. Se observa un portal azul sobre la plataforma naranja conectado a un portal rojo sobre la plataforma cian. Una pared gris separa ambas plataformas.

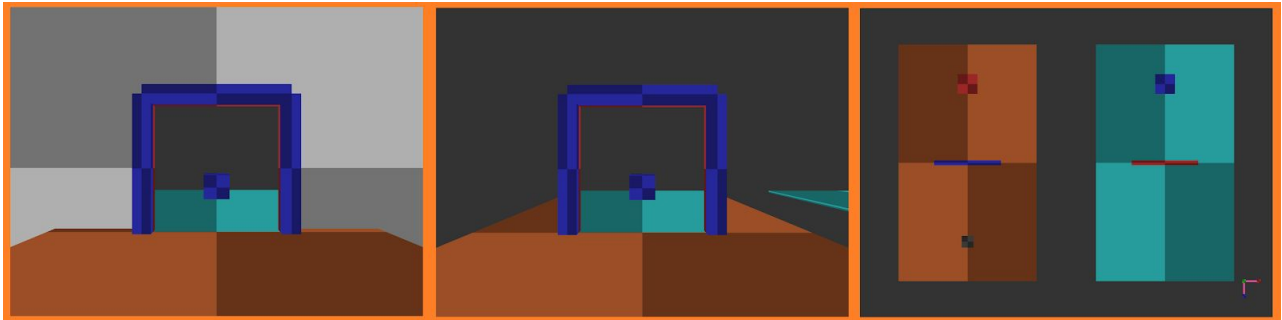
- En el punto de vista **[A]** (cenital en ortogonal), se muestra como están conectados los portales; dos flechas pequeñas muestran la dirección de cada portal, que importa porque al atravesar una cara, se sale por la contraria del otro portal.
- En el punto de vista **[B]** (primera persona en perspectiva), se muestra como la luz atraviesa el portal (imagen del cubo sobre la otra plataforma). Este efecto permite ver al otro lado del portal contrario y también se puede atravesar caminando.
- En el punto de vista **[C]** (alejado en ortogonal), se muestra la escena completa con los portales desactivados.

Se considera una *discontinuidad* ya que implica que no se desplace ninguna geometría, y esto de alguna forma, altera el curso natural del espacio euclidiano. Ya no se puede aceptar el primer y más básico postulado de **Euclides**: “Existe exactamente una única línea recta que pasa entre dos puntos” (y el camino más corto entre dichos puntos está contenido en dicha recta). Debido a los portales, entre dos puntos puede existir más de una recta que pasa entre ellos (y el camino más corto puede no ser el original). Pero inevitablemente, además de crear nuevas rectas entre dos puntos determinados, el propio portal también puede bloquear otras existentes. Esto ocurre debido a que la superficie del portal ejerce de *barrera*; por ejemplo, una trayectoria recta que pasa por donde se encuentra el portal, debe en su lugar rodearlo si no quiere desplazarse a través y terminar al otro lado del portal conectado.



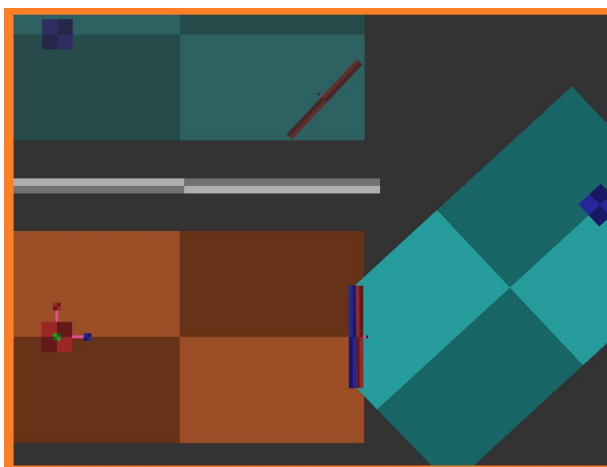
[Figura 5.1.2]: Esquema con las mismas plataformas, ahora sin estar separadas por una pared. En esta vista cenital se muestra super-posicionada la plataforma azul para representar la geometría visible observada a través del portal desde la plataforma naranja. Originalmente, la recta horizontal amarilla es la única que contiene un camino $[P \rightarrow X]$. Pero debido a los portales, existe otra recta vertical que contiene también $[P \rightarrow X]$, además más corto. El camino recto $[X \rightarrow I]$, se ve bloqueado debido al portal rojo, ahora su trayectoria continúa como $[X \rightarrow II]$. Para realizar un camino $[X \rightarrow I]$, se debe rodear el portal rojo.

Otra forma de comprender un portal es visualizarlo como una apertura entre dos estancias, solo que en realidad éstas no son tangentes en el espacio tridimensional. La transición es totalmente inapreciable, como si ambos lugares estuviesen simplemente conectados. Esto no implica que un par de portales deban estar siempre sobre la superficie de paredes, pero en esos casos si se añade un marco, no existe forma obvia de notar la diferencia respecto a una simple apertura real en la pared. Podrían estar separados por apenas unos centímetros o cientos de metros.



[Figura 5.1.3]: Capturas de pantalla con una vista en primera persona (en perspectiva). A la izquierda, se muestra como un portal sobrepuesto a una pared es indistinguible de una puerta entre dos estancias. En este caso las estancias son la plataforma naranja y la cian. En el centro se muestra un portal bidireccional, simplemente sin pared que bloquee uno de sus lados. Se añade una pequeña vista cenital como mapa de la escena.

Si se limita la simulación a una única perspectiva y con portales situados en estancias separadas (que desde una no se vea la otra), es posible crear el mismo efecto descrito simplemente desplazando la geometría de la forma adecuada. Específicamente, se debería transformar toda la escena fijando las superficies de ambos portales como coplanares. Pero este enfoque no es del todo práctico ya que a pesar de sus severas limitaciones, mantiene igualmente cierto nivel de complejidad.



[Figura 5.1.4]: Misma escena que en la figura [5.1.2], pero se ha añadido la pared separadora. En este caso la plataforma azul se ha trasladado realmente (se observa semitransparente su posición original). Este método no funciona si se elimina la pared, ya que entonces se ve que la plataforma original ha desaparecido. Y obviamente, cualquier perspectiva externa también es capaz de ver el fallo.

5.2. Caso de estudio

Existen múltiples ejemplos en los que se han implementado portales con las propiedades descritas. En algunas ocasiones, no se interactúa con ellos, por lo que sólo son visuales; o son sutiles y no se percibe su uso. Estos ejemplos son producciones de todo tipo de industrias: animación, cine, videojuegos, etc. Este proyecto se centra en los videojuegos ya que son lo más cercano al objetivo. Mientras que en cine un portal puede ser retoque fotográfico, en un videojuego se debe simular de forma interactiva a tiempo real.

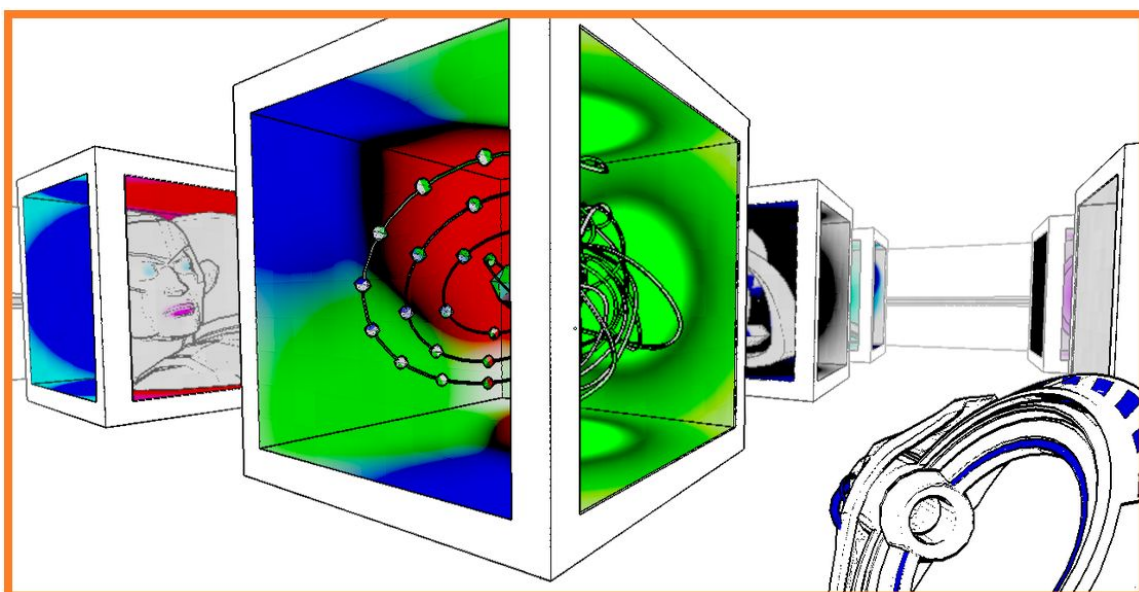
El caso de estudio de este proyecto es *Portal* [3], desarrollado por *Valve Corporation* [56] y publicado en 2006. Sin lugar a dudas, es el videojuego que implementa portales más famoso y exitoso comercialmente. Es un juego de puzzles en primera persona, donde el jugador debe resolver los problemas que se le plantean utilizando un dispositivo capaz de crear portales a distancia, sobre determinadas superficies. La ambientación consiste en un futuro distópico en el que una IA obliga a un sujeto a realizar pruebas peligrosas (*dichos puzzles*). El juego entero está diseñado alrededor de portales y obviamente es totalmente explícito respecto a su uso.

Tiene una secuela, *Portal 2* [57], lanzada en 2011. Ambos juegos han sido aclamados por la crítica y han vendido múltiples millones de copias, tanto físicas como digitales. Parte del equipo de desarrollo de *Portal*, realizó previamente un videojuego más pequeño, *Narbacular Drop* [58]. Este juego se considera su antecesor, fue desarrollado en 2004-2005 como proyecto final de estudios en *DigiPen Institute of Technology* [59] (USA). *Valve* se interesó por el proyecto y contrató al equipo entero para que desarrollara *Portal*.



[Figura 5.2.1]: A la izquierda, una captura de *Portal* [3], se observa un puzle sencillo en el que se utiliza un portal para superar un salto infranqueable. A la derecha, una captura de *Narbacular Drop* [58], se observa dos portales conectados.

También existen ejemplos de videojuegos que utilizan portales diferentes. Un caso interesante es *Antichamber* [60] (2003), nuevamente en primera persona y centrado en puzles. Este juego utiliza portales de forma más sutil: crea situaciones y geometrías no euclidianas ocultando sus límites. Contrastando con otros usos creativos de portales, se pueden encontrar casos en los que son necesarios por cuestiones técnicas; por ejemplo, dos habitaciones separadas debido a limitaciones espaciales o de *nivel*, es posible conectarlas con una puerta *falsa* basada en un portal inapreciable.



[Figura 5.2.2]: Captura de pantalla de *Antichamber* [60], se muestra una sala llena de cubos cuyas caras permiten ver en su interior escenas distintas.

5.3. Fuentes de información

La popularidad de los videojuegos de *Portal* en sí misma, no es la razón por la que componen el caso de estudio del proyecto. Es la destacable información técnica disponible sobre ellos, la que los hace tan valiosos. Se puede pensar que esta colección de información se debe a su éxito, pero no necesariamente, ya que los mejores contenidos técnicos son los publicados por los desarrolladores en charlas, y no los que se encuentran recopilados o especulados en wikis.

Cómo se concreta a continuación, la mejor fuente de información es reciente, data de 2018, y otra muy buena pero antigua, proviene del antecesor *Narbacular Drop*. También se pueden encontrar distintos recursos en los que se recrea portales sobre diferentes arquitecturas, la mayoría motores modernos (*Unity* o *Unreal*). Estas implementaciones suelen seguir las ideas y metodologías asentadas en *Portal* y su antecesor; principalmente utilizan técnicas menos sofisticadas, pero los fundamentos siguen siendo interesantes. Este proyecto utiliza las siguientes fuentes relacionadas con la saga de *Portal*:

- Documentos oficiales del juego *Narbacular Drop* [61]. Incluidos el de diseño de juego, diseño técnico y postmortem. Siendo de gran interés el de diseño técnico [62] donde, entre otras cosas, se describe la API gráfica empleada y brevemente un esquema con el proceso utilizado para implementar los portales. Aunque son pocos datos, permiten contrastar otras fuentes de información.
- Grabaciones de la conferencia dada en 2018 en Harvard para el curso *CS50* [63] por dos empleados de *Valve*: *Dave Kircher* (partícipe en el desarrollo de todos los juegos de la saga) y *Tejeev Kohli* (miembro del equipo de *Portal 2*). En esta charla se comparan las diferentes tecnologías y metodologías utilizadas en los juegos de *Portal* y su antecesor. Se destacan problemas que surgen durante el desarrollo y las soluciones planteadas. Estos problemas son de todo tipo (diseño de niveles, renderizado, física, etc), los más relevantes son aquellos relacionados con la informática gráfica. Este recurso es la mejor fuente de información disponible con diferencia y todo su contenido está bajo una licencia permisiva (*CC BY-NC-SA 4.0* [64]).
- Grabaciones de conferencias sobre *Portal 1* y *2* en distintas ferias de videojuegos [65] como la GDC. La mayor parte del contenido es sobre el diseño y el argumento, por lo que resultan de menor interés.
- También existen wikis [66], documentales [67], reportajes y entrevistas... profesionales y/o realizados por fans; pero de nuevo, la mayor parte no contiene nada técnico relevante que no estuviese presente en las fuentes de información anteriores.

What is a Portal?



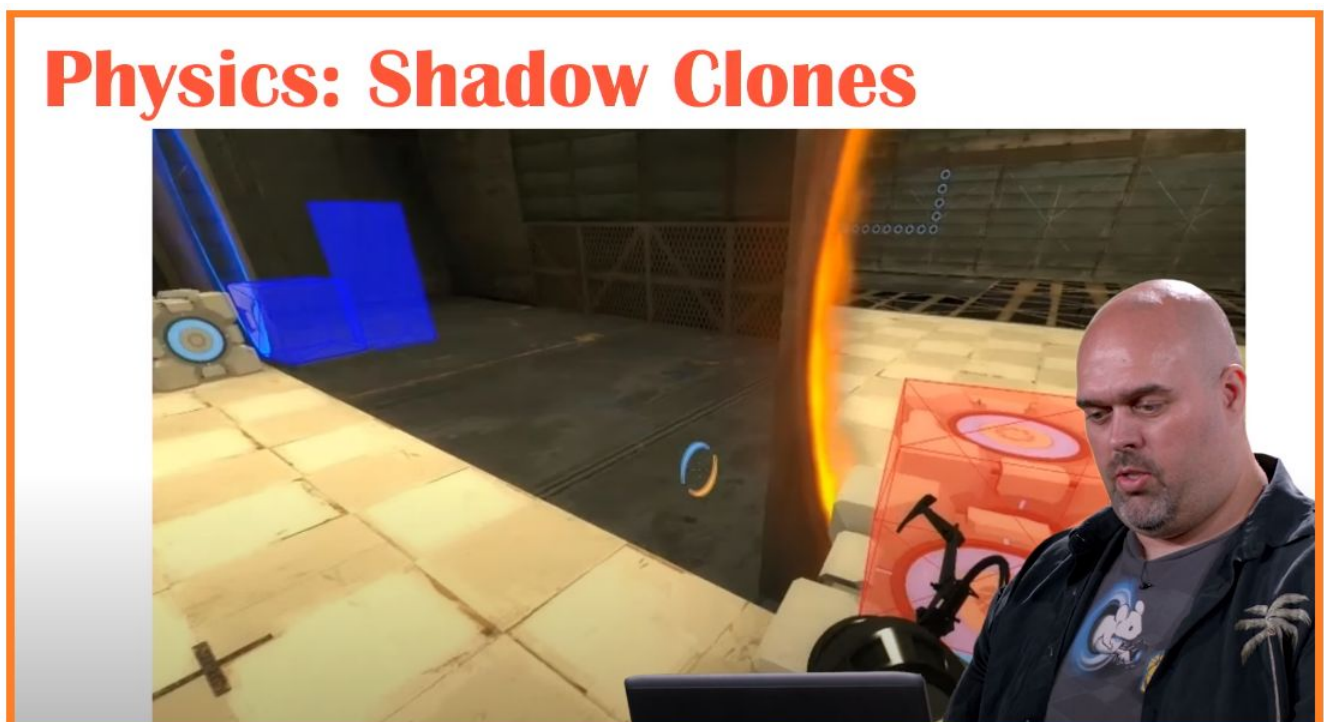
[Figura 5.3]: Captura de pantalla de parte de la conferencia dada por Dave Kircher para CS50 [63]. Se muestra una compuerta en un nivel de Portal 2 que, por razones técnicas, debía ser un portal inapreciable para el jugador. Al estar ambos portales sobre la superficie de paredes, es indistinguible a una simple puerta.

5.4. Métodos de implementación de portales

Las características de un portal se pueden separar a la hora de implementar su funcionamiento. Una parte es la visual, la representación de una escena que se observa correctamente a través de la superficie del portal. La otra parte es la física; el transporte de objetos que atraviesan el plano del portal. Ambas partes pueden implementarse partiendo de ideas alternativas, con sus ventajas y desventajas.

Existen multitud de detalles a tener en cuenta y todos son relevantes; al fin y al cabo, la parte más interesante de los portales es que el usuario no note ninguna transición, y cualquier mínimo fallo puede estropearla. Además es importante prevenir errores, ya que en caso de tener lugar, se puede producir desorientación debida a artefactos visuales, comportamientos físicos no anticipados, situaciones carentes de sentido, etc. Como un truco de magia, debe parecer simple, ocultar todo mecanismo complejo y no mostrar nada más que la *superficie*.

La implementación de la parte física del funcionamiento de los portales, varía sobre todo según la precisión y realismo deseado. No es lo mismo transportar un único objeto y conservar sus velocidades, que mantener interacciones correctas entre objetos, incluso si sólo atraviesan un portal parcialmente. La simulación de otros fenómenos físicos como luces o el sonido también es posible, pero no se suele plantear realmente porque su efecto pasa desapercibido en la escena. En este proyecto se prioriza la parte gráfica sobre la física (no se pretende crear un motor de físicas); por ello se limita simplemente a desplazar el jugador, es más, no se calculan colisiones entre ningún elemento.



[Figura 5.4.1]: Captura de pantalla de parte de la conferencia dada por Dave Kircher para CS50 [63]. Se muestra la compleja simulación física que tiene lugar entre los objetos cercanos a un portal. Se utilizan listas de colisiones y clones invisibles al otro lado de los portales para lograr choques realistas.

Respecto a la parte gráfica, existen varios métodos de implementación, estos se fundamentan en diferentes conceptos de la informática gráfica explicados en el capítulo [4] de la memoria. De nuevo, estos conceptos están presentes en las diferentes API gráficas aunque sea con otros nombres. Se ha de destacar, como se comenta posteriormente, que no todos los procesos son soportados por cualquier versión de API.

- Mediante **texturas enlazadas a un *framebuffer***: Este método consiste en asignar la textura de cada portal a un *framebuffer*. Primero se activa cada *framebuffer* para escribir en cada una de estas texturas la imagen de la escena, desde el punto de vista adecuado para cada portal. Posteriormente, se activa el *framebuffer* de la pantalla y se dibuja toda la escena vista a través de la cámara principal; en dicha escena aparecen los portales utilizando las texturas previamente calculadas.
- Mediante ***stencil-buffer***: Este método consiste en el uso del *stencil-buffer* para descartar fragmentos fuera de la superficie del portal. A diferencia de en el método anterior, primero se dibuja la escena vista a través de la cámara principal, sin portales. Después se dibuja cada superficie de portal con el *stencil-buffer* activado, para que queden marcados sus fragmentos. Finalmente se dibuja la escena, desde el punto de vista adecuado para cada portal, pero se descarta todo fragmento cuya posición no estuviese marcada previamente en el *stencil-buffer*.
- El último método está relacionado con técnicas de **renderizado físico**. Estas técnicas tratan de renderizar el mundo de una manera más realista; se aproximan al comportamiento de la luz, simulando rayos y rebotes (en inglés, conocido como “*raytracing*”). La gran ventaja que aporta es la iluminación y reflejos generales sin importar las condiciones, pero esto tiene un alto coste computacional, por ello no resulta útil para situaciones que requieren interactividad a tiempo real. Hoy en día se desarrollan cada vez mejores tarjetas gráficas con hardware específico para este tipo de renderizado (RTX), para poco a poco aliviar dicha carga computacional del software. Para simular portales en esta tecnología, habría que interceptar los rayos que choquen contra la superficie de un portal, y generar su salida (con la dirección adecuada) por la superficie contraria de su portal conectado.



[Figura 4.4.2]: (Imagen de “*Learn OpenGL*” [24]). Ejemplo de renderizado basado en física (conocido por sus siglas en inglés, “PBR”). Los materiales resultantes son espectaculares, pero tiene un alto coste de computación.

5.5. Comparativa de métodos

Respecto a las funcionalidades necesarias de las API, el uso de texturas enlazadas a un *framebuffer*, está soportado prácticamente desde las primeras versiones de API y tarjetas gráficas. Hoy en día los *stencil-buffer* también, pero su introducción fue posterior. En cuanto al renderizado físico, se necesita hardware relativamente moderno para que sea viable a tiempo real (en escenas complejas).

Retomando el ejemplo del caso de estudio, *Narbacular Drop* utiliza exclusivamente texturas para renderizar los portales. En *Portal* (el primero), se comienza a usar la técnica que implica el *stencil-buffer*, pero se mantiene la implementación con texturas disponible como opción para tarjetas antiguas de la época. *Portal 2*, publicado 5 años después, utiliza directamente *stencil-buffer* de forma exclusiva.

Portal Effect

The creation of our portals takes place with some extremely simple trickery along with a lot of tedious things to make it work well.

Essentially, a portal is just a textured quad. But the contents of the texture are dynamic and the texture coordinates change based on how you're looking at the quad.

To create the contents of the texture:

1. Convert the current camera position and look vector into coordinates relative to the portal.
2. Find this position and rotation in relation to the portal's exit.
3. Move the camera to the point and rotation found in step 2.
4. Render the scene while culling all objects between the camera and the portal exit.
5. The render surface is now the source texture for the portal and the camera should be moved back to where it was before step 1.

To render the quad with the proper texture coordinates:

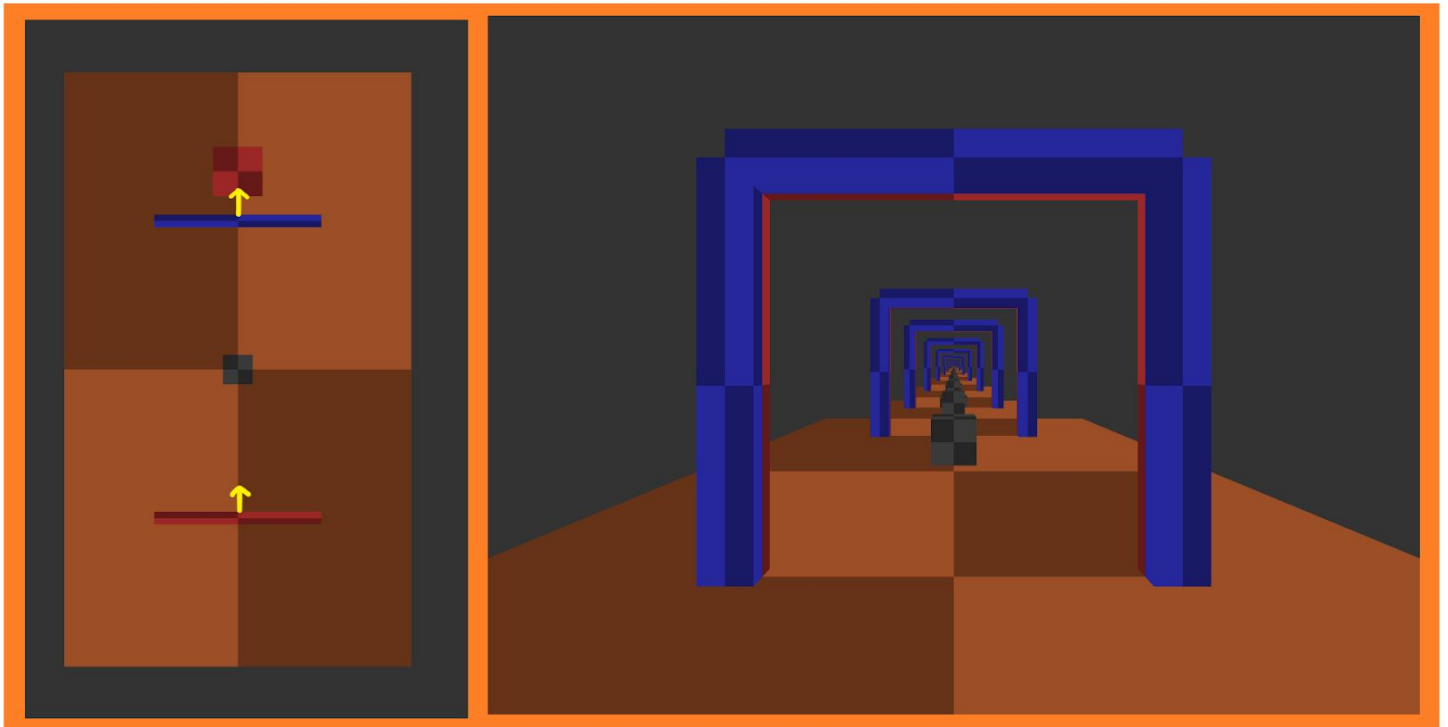
1. Convert each coordinate into screen space.
2. Copy the normalized screen space x and y components into the u and v coordinates of each vertex.
3. Render the quad.

[Figura 5.5.1]: Extracto del documento de diseño técnico de *Narbacular Drop* [62], en este fragmento se describe el método utilizado para implementar sus portales. Traducido del inglés “Esencialmente, un portal es simplemente un rectángulo texturizado. Pero los contenidos de la textura son dinámicos y sus coordenadas de textura cambian en base a como se mira el rectángulo”. Como curiosidad, la API utilizada es Direct3D en C++ y se puede ver como se habla de conceptos muy parecidos.

Desde un punto de vista pragmático, no tiene mucho sentido comparar la técnica que implica renderizado físico con las otras dos, ya que es fundamentalmente diferente. En su caso, se simplifica el proceso lógico, por el coste de cómputo que implica simular de forma realista el comportamiento físico de luz en la escena. Este tipo de renderizado se aleja de los conceptos básicos de la informática gráfica explicados en este proyecto, por ello se descarta y se centra la atención a los otros dos métodos.

La principal ventaja que aporta realizar los portales mediante el método con texturas, es la abstracción que proporciona. Los portales sencillamente se pueden ver como objetos normales con una determinada textura, la cual tiene la peculiaridad de que ha de ser pintada previamente. Esa abstracción permite separar el renderizado de la escena y el de las texturas individuales de cada portal. Por otro lado tiene sus repercusiones:

- Se necesita utilizar memoria para *framebuffers* auxiliares para pintar cada textura (separados del *back-buffer* por defecto de pantalla). Se puede utilizar uno único, si se pintan los portales secuencialmente.
- Aunque no es obvio, que un portal se simule como una superficie plana puede provocar artefactos visuales inesperados. Por ejemplo, al acercarse la cámara, su plano cercano puede atravesar la superficie del portal y esto en la vista final provocaría ver fragmentos que se encuentran detrás del mismo. También se complican efectos de post-procesado de escena que necesiten información de profundidad, ya que ésta directamente no es la real.
- En caso de que se vea un portal a través de otro, se necesita lógica para poder pintar primero la textura del portal situado en el fondo. Esto se debe a que dicho portal y su textura aparecen posteriormente en la textura del portal en primer plano (o simplemente delante). En caso de recursión ocurre igual, es necesario dibujar la más profunda primero, y ello suele implicar la necesidad de calcular y almacenar todas las posiciones previas. La recursión en esta situación, hace alusión a cuando a través de un portal **A** se ve otro **B** y en la textura de este **B** además se ve de nuevo **A** y así continúa recursivamente.



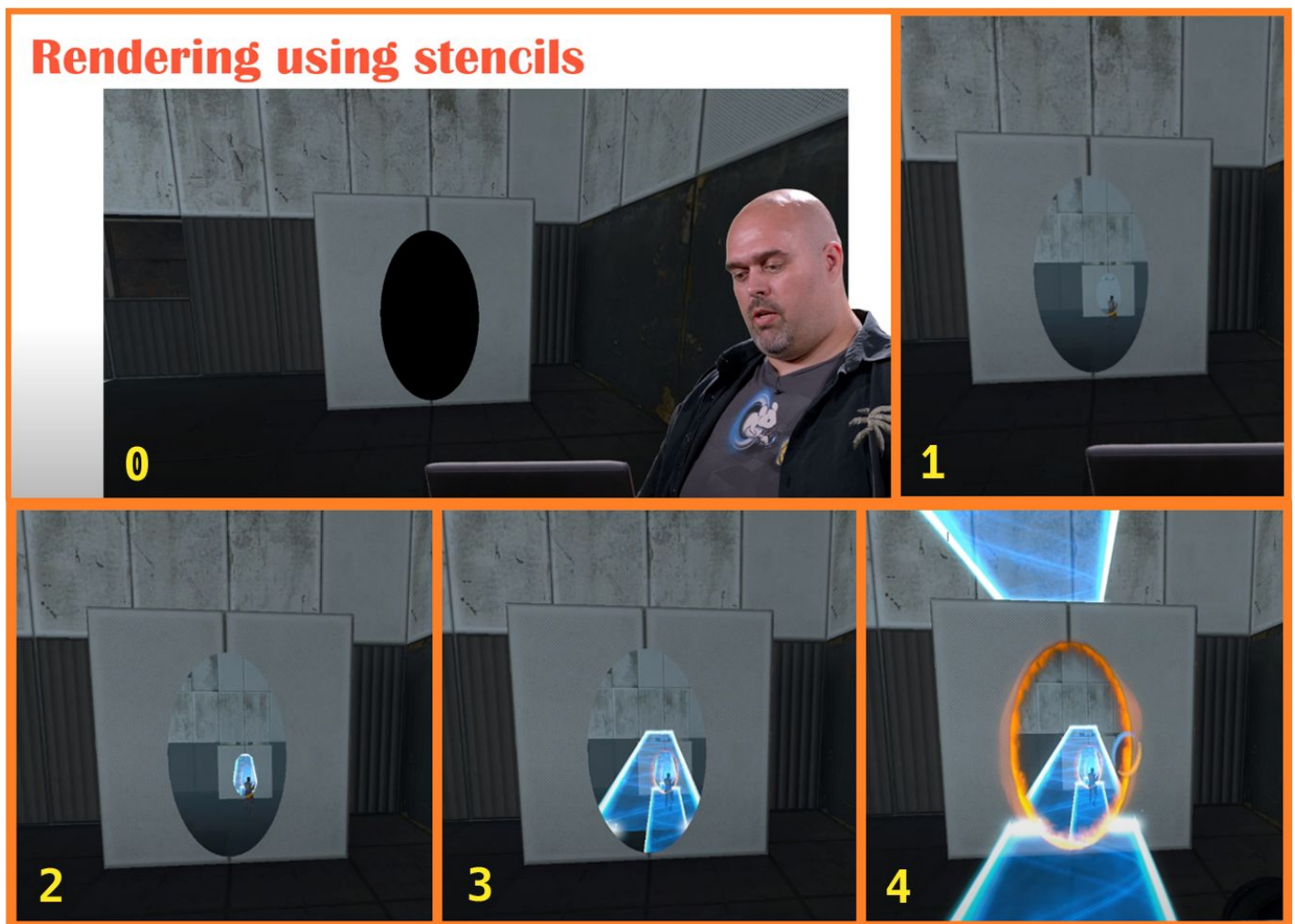
[Figura 5.5.2]: *Captura de pantalla con una vista en primera persona (en perspectiva) a la derecha y una vista cenital ortogonal como mapa de la escena a la izquierda. La recursión es uno de los mecanismos más complejos de los portales. Para que ocurra simplemente un portal debe estar detrás de otro (ambos orientados en el mismo sentido). Pueden estar ligeramente desalineados o rotados y esto provoca efectos drásticos en la recursión.*

Curiosamente, el beneficio más interesante que se obtiene al realizar los portales mediante el método con *stencil-buffer*, es que no surgen los inconvenientes que se producen con el método anterior:

- No se necesita memoria adicional para *framebuffers*, todo se escribe directamente en el mismo *back-buffer* por defecto de pantalla (utilizando un *stencil-buffer*).
- La profundidad de las escenas vistas a través del portal es real, con ello se consigue una calidad de imagen homogénea en los efectos de post-procesado comentados, sin mayor esfuerzo. También se evitan problemas de cámara con la superficie del portal, aunque persisten en caso de estar sobre la superficie de una pared.

- Se dibuja directamente la escena desde el punto de vista de la cámara principal. Posteriormente, se utiliza siempre la misma lógica para dibujar los portales, independientemente de su posición relativa. En caso de recursión se dibuja de menor a mayor profundidad, no es necesario calcular posiciones previamente y además se puede detener con facilidad a un determinado nivel.

Por desgracia, al utilizar *stencil-buffer* se producen otros inconvenientes. Esencialmente requiere una lógica bastante más compleja, se pierde la sencillez y la abstracción que aporta el método basado en texturas; en caso de usar *stencil-buffer*, se requiere un orden de renderizado complicado que entrelaza los tipos de objetos de la escena y los niveles de recursión. Por ejemplo, se empieza por renderizar los objetos opacos desde el punto de vista principal y después los opacos de cada recursión de cada portal hasta la última; posteriormente se continúa con los objetos translúcidos desde la última recursión de cada portal, hasta la primera y el punto de vista principal.



[Figura 5.5.3]: Capturas de pantalla de parte de la conferencia dada por Dave Kircher para CS50 [63]. En la primera imagen [0] se muestra el mencionado paso intermedio en el que se pinta la superficie del portal dejando marcado el stencil-buffer para posteriormente descartar todo fragmento cuya posición no estuviese marcada.

1. Se comienza dibujando todos los objetos opacos hasta el final de la recursión.
2. Se continúa dibujando los objetos translúcidos desde la última recursión.
3. Ya se han dibujado los translúcidos de la primera recursión.
4. Finalmente se dibujan los translúcidos desde el punto de vista principal.

5.6. Conclusión

En este capítulo se concreta la definición de portal, se habla sobre los casos de estudio interesantes y las fuentes de información disponibles destacadas. Posteriormente se explican los diferentes métodos de implementación y se realiza una comparación entre ellos. Ahora para poder concluir, sólo falta especificar el método utilizado en este proyecto y el razonamiento detrás de esta decisión

Se han desarrollado los portales utilizando **texturas enlazadas a framebuffer**s; su ventaja principal, la separación del renderizado de la escena y de las texturas de cada portal, permite realizar una explicación más ordenada y de mayor calidad. Todos los cálculos y matemáticas requeridos para su funcionamiento son comunes, lo que varía es el uso final de la API para dibujarlos. El rendimiento de los dos métodos es parecido, en general el mayor impacto lo tienen situaciones recursivas que se pueden dar en ambos casos.

El método escogido proporciona una experiencia más didáctica desde el punto de vista de la informática gráfica. En su caso, los problemas que surgen, se relacionan y se resuelven con conceptos de la misma. Por otro lado, en el método basado en *stencil-buffer*, los problemas son esencialmente de gestión lógica general y orden de renderizado complejo. Luego se ha optado por utilizar texturas, debido a que una parte importante de este proyecto es la propia explicación sobre la implementación de los portales, por ello se prefiere la técnica que utiliza más conceptos relacionados con la informática gráfica y se ven las dificultades que puedan surgir como oportunidades de expandir sobre conceptos de ésta. En general, el método basado en texturas permite realizar una mejor explicación sobre los portales, y además, una vez comprendidos sus fundamentos y su implementación gráfica, es más fácil emprender otro desarrollo con *stencil-buffer*.

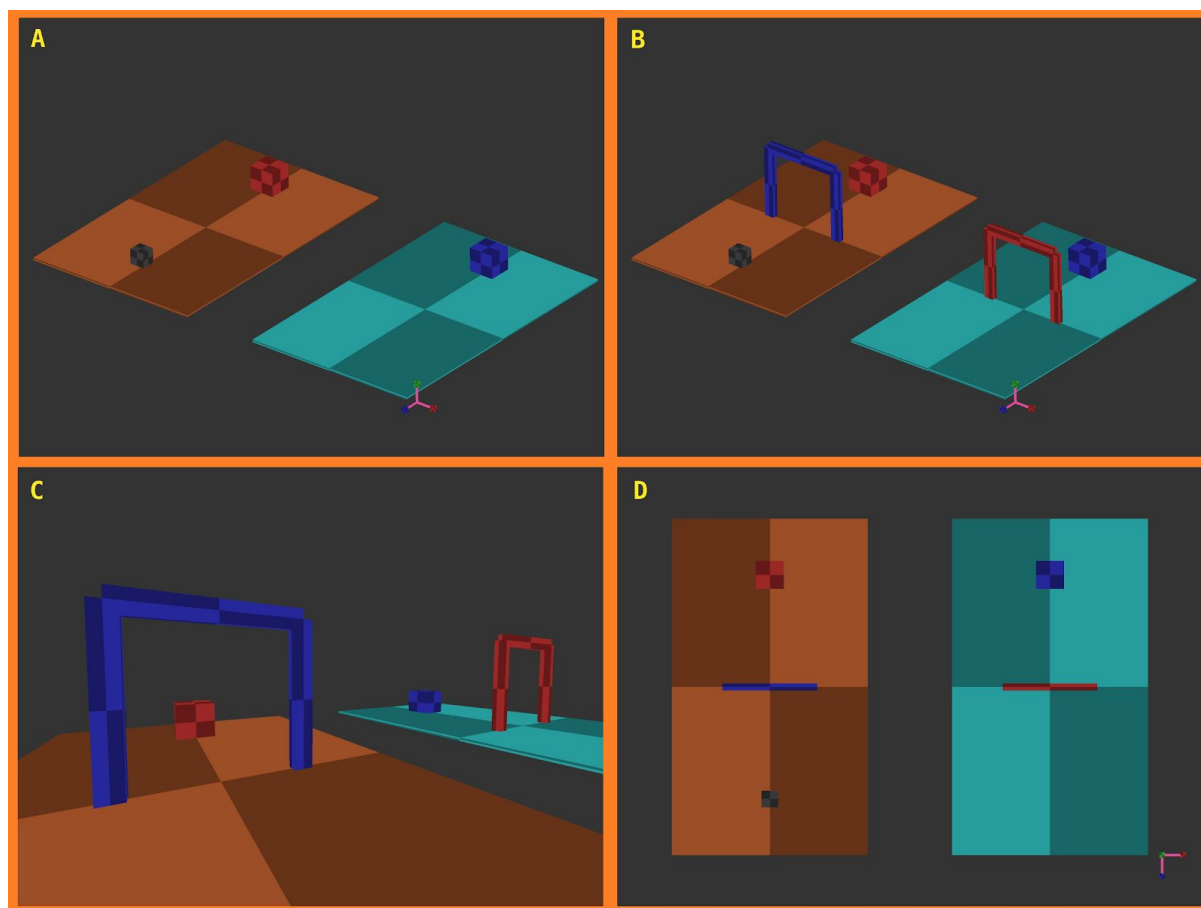
6. Arquitectura de la aplicación: *Portales*

Ha llegado el momento de la implementación de los portales, de detallar paso a paso su funcionamiento. No se desarrolla ni explica directamente un portal que cumpla todas las características especificadas de golpe; es un proceso complicado, por ello lo ideal es abordarlo de forma incremental. Por ejemplo, un buen objetivo inicial es lograr un portal que permita ver a través de su superficie correctamente, es decir que muestre la escena que se encuentra al otro lado del portal contrario. Una vez obtenida una implementación funcional que cumpla dicha propiedad, se puede proceder a construir sobre ella otras (atravesar físicamente portales, solucionar artefactos visuales en situaciones extremas, recursión, etc). Se añade pieza por pieza, siempre con la idea de obtener nuevas versiones que funcionan adecuadamente por sí mismas, y que cada vez soportan más características propias de un portal. Se sigue esta metodología de desarrollo iterativa e incremental hasta conseguir la implementación suficientemente completa de un portal.

Antes de comenzar se necesita identificar las partes más básicas por las que se debe empezar a trabajar. Igualmente se tienen que tener en cuenta las posibles dependencias a la hora de planear el orden de desarrollo. Se debe tener cierto cuidado, pero siempre es posible volver a una versión anterior para realizar una implementación alternativa. El proceso de desarrollo termina cuando se satisface el nivel de detalle deseado; por ejemplo, este proyecto se centra en el apartado visual de los portales y se despreocupa del comportamiento físico. La explicación que sigue a continuación, también se realiza de forma análoga a esta metodología (esto es común ya que el conocimiento se suele exponer de forma incremental). Concretamente, tanto la implementación como la explicación se han dividido en la siguiente lista de hitos. Durante los mismos, también se exponen los detalles teóricos y los problemas que pueden surgir, junto con sus posibles soluciones.

1. Vista a través de la superficie de portales.
2. Desplazamiento en primera persona a través de portales.
3. Vista externa de un desplazamiento a través de portales.
4. Vista recursiva a través de portales.

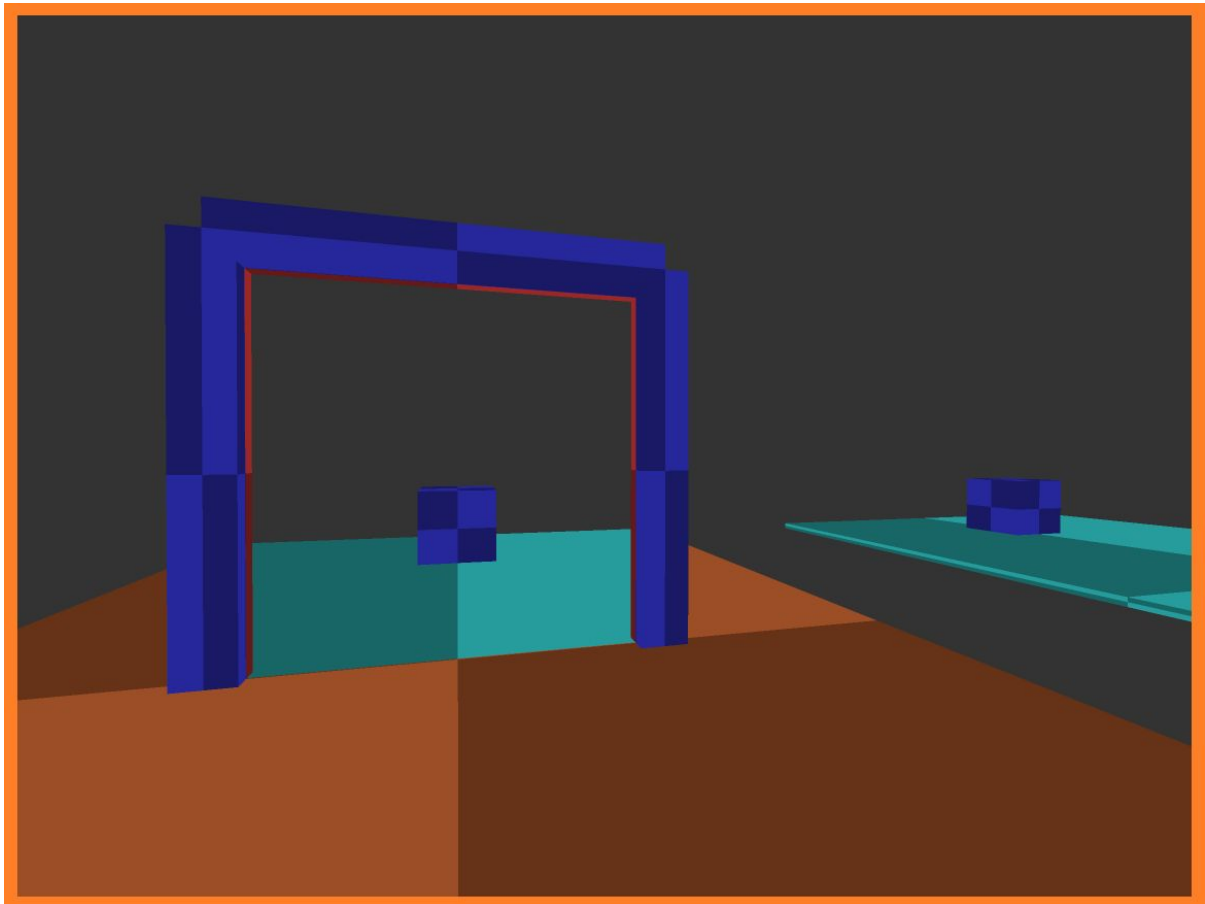
Todo el desarrollo se hace sobre la arquitectura de la aplicación previamente creada. Se cuenta con sus capacidades como motor gráfico y sus escenas estructuradas en árboles de nodos, cada uno con una transformación asociada. Además, de estos se pueden derivar otros objetos para ejecutar comportamientos lógicos diferentes; incluido leer datos del estado de la plataforma (entrada, tiempos, tamaño de ventana, etc). La implementación comienza a partir de una escena sencilla (mostrada en la figura [6.0]) y un sistema de control que permite moverse por la misma.



[Figura 6.0]: Esquema con varios puntos de vista de la escena inicial. En [A] se observa un pequeño cubo oscuro que representa el cuerpo del jugador que se mueve en primera persona, la cámara se posiciona directamente encima. También se observan dos plataformas con un cubo cada una. El código de colores y las texturas sencillas ayudan a comprender mejor la escena, las perspectivas y la disposición espacial de los objetos, que en ocasiones se pueden complicar debido a los portales. En [B] se han posicionado los marcos que representan los límites de cada portal; son del mismo color que el cubo que se debe ver a través de ellos cuando existan portales. En [C] se muestra una vista en primera persona (desde el pequeño cubo oscuro), en la que se puede todavía no hay ningún portal activado y son sólo simples marcos. En [D] se complementa el esquema de la escena con una vista cenital ortogonal.

Los siguientes apartados de este capítulo, siempre siguen la misma estructura: comienzan con una imagen que muestra el objetivo a cumplir en dicho apartado, y posteriormente se procede a explicar paso a paso el funcionamiento y las técnicas empleados para lograrlo.

6.1. Vista a través de la superficie

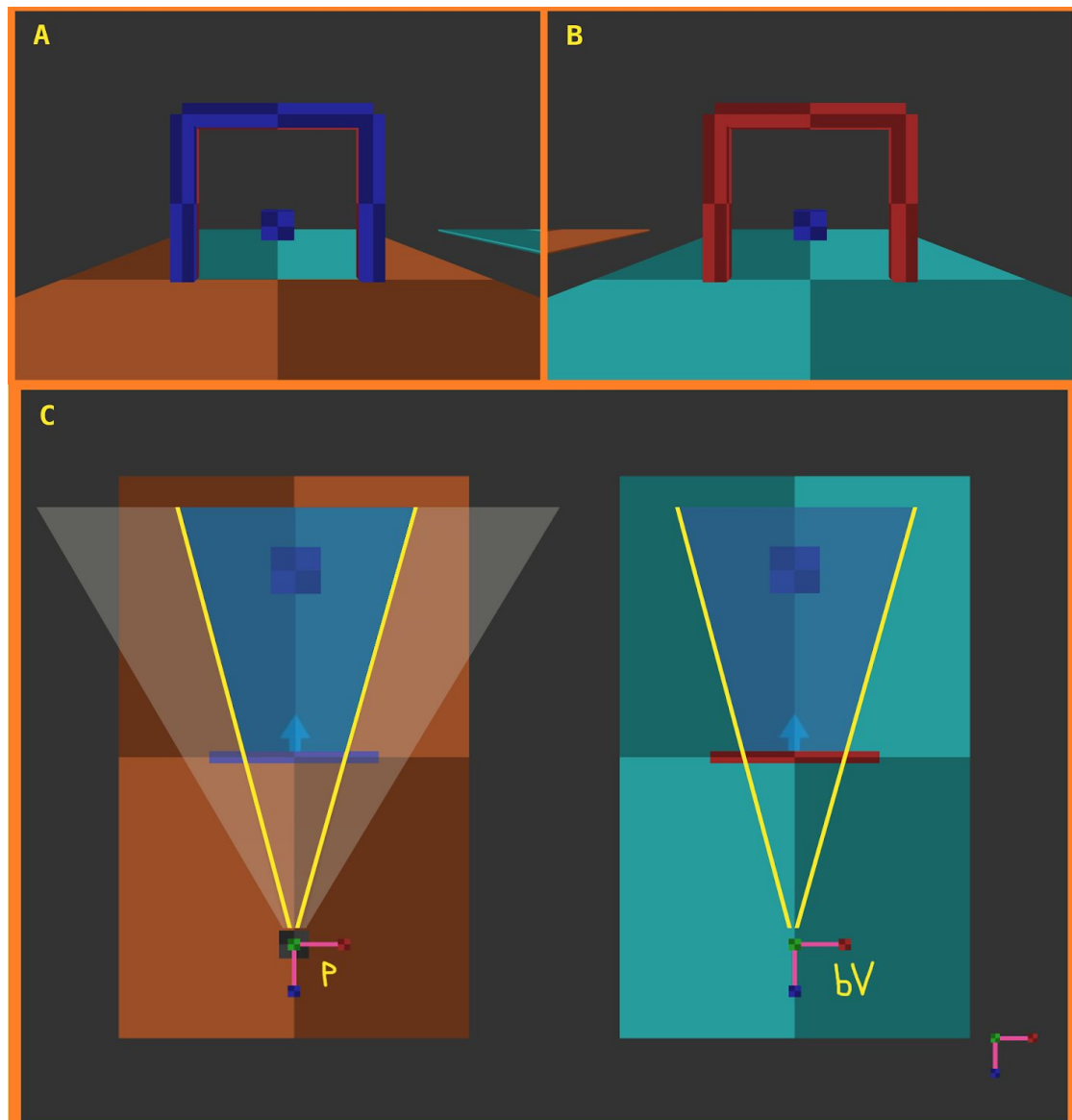


[Figura 6.1.1]: *Vista en primera persona (en perspectiva), en la que se ve satisfactoriamente a través de la superficie del portal.*

Inicialmente, se reduce el planteamiento a un único portal y posteriormente se extiende para poder observar ambos simultáneamente en pantalla. El primer paso es entender que la imagen mostrada en la figura [6.1.1] se compone de dos vistas (ambas en primera persona) separadas y es necesario identificarlas:

- La parte de la escena observada *fuera* del portal, proviene del punto de vista original. En caso de no existir el portal, no cambia.
- La parte de la escena observada *dentro* del portal, proviene de otro punto de vista. En caso de no existir el portal, no se ve; se vería la misma plataforma y el cubo rojo.

Un punto de vista depende de la transformación de la cámara y su configuración, luego como la configuración es la misma, lo que se debe encontrar es la transformación adecuada (traslación, rotación y escala). De momento la escala se puede ignorar, ya que ambos portales forman parte de la misma escena y todos los objetos están a la misma escala. Independientemente, como en la implementación se trabaja con las transformaciones completas (matrices de modelado), cualquier modificación de la escala también funciona sin problema. Para poder referirse a este punto de vista importante con mayor facilidad, se le denomina como una cámara *virtual*; entonces en la figura [6.1.1] se muestra una composición de los puntos de vista de la cámara principal y la cámara virtual del portal azul. No es necesario que sea un *objeto* cámara independiente, pues basta con guardar la transformación que habilita dicho punto de vista, pero por simplicidad, en este proyecto se utiliza directamente *objetos* cámara adicionales (que se representan en las capturas de la escena).



[Figura 6.1.2]: Esquema con varios puntos de vista de la escena inicial con un portal funcional. En [A] se observa el punto de vista de la cámara principal, en el que se ve parte de la plataforma cian a través del portal. En [B] se observa el punto de vista de la cámara virtual, en el que simplemente se ven los objetos de la plataforma cian. En [C] se muestra una vista superior de la escena con un esquema (no a escala) del espacio de vista de la cámara principal [P de “player”] superpuesto. Como se puede ver, parte de la superficie de dicho espacio se encuentra al otro lado del portal, y lo que ocurre es que se reemplaza por exactamente el mismo espacio pero visto por la cámara virtual [bV de “blue virtual”].

La posición de la cámara virtual ha de ser dinámica, depende de la posición de ambos portales y del jugador. En este caso, la posición de la **cámara virtual azul** en coordenadas *locales* del portal **rojo**, siempre debe ser igual a la posición del **jugador** (cámara principal) en coordenadas *locales* del portal **azul**. Cumpliéndose esto se asegura que aunque se desplace el jugador o cualquiera de los portales, su posición se actualiza correctamente. Pero no se trata sólo de posición (3D), también de su rotación y escala. Este proceso se consigue utilizando las matrices de modelado, que como ya ha sido explicado, representan la transformación completa de un objeto. Específicamente, también se necesita la matriz de modelado inversa para poder obtener cualquier transformación en el sistema de coordenadas de otro objeto.

```
// CASE A: blue portal cam is already a child of red portal
// (therefore we just need red portal local coordinates)

// blue VIRTUAL CAMERA Model Matrix = bluePortal Inversed M.M. * mainCam M.M.
// blueCam [T] = bluePortal [iT] * mainCam [T]
bPortalCam_>setLocalTrans(bPortalRoot_>getModelMatrix_Inversed() * cam_>getModelMatrix());

// CASE B: blue portal cam is a global scene object
// (add transformation to world from red portal local)

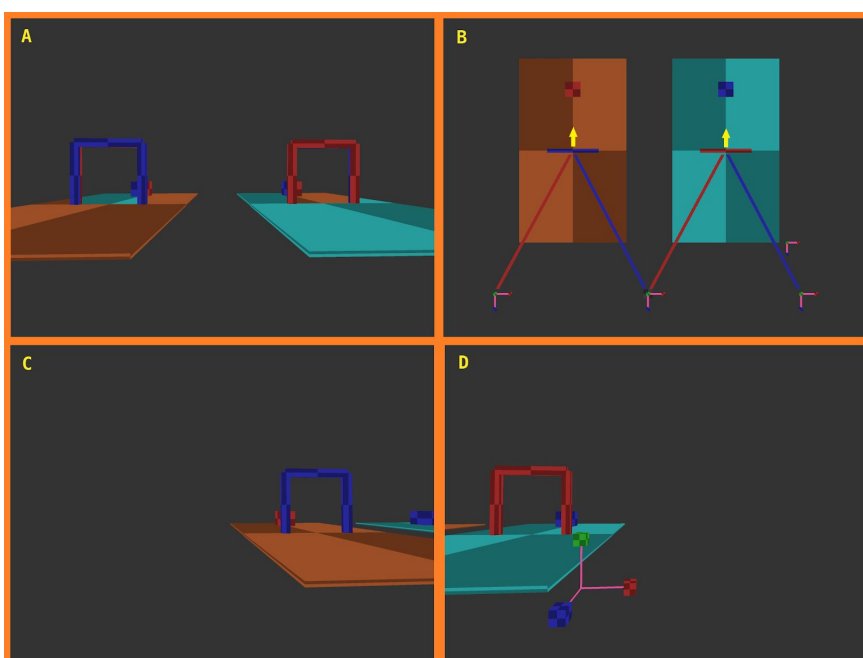
// blue VIRTUAL CAMERA Model Matrix = redPortal M.M. * bluePortal Inversed M.M. * mainCam M.M.
// blueCam [T] = redPortal [T] * bluePortal [iT] * mainCam [T]
bPortalCam_>setLocalTrans(
    rPortalRoot_>getModelMatrix() * bPortalRoot_>getModelMatrix_Inversed() * cam_>getModelMatrix());
```

[Figura 6.1.3]: Muestra de las fórmulas del producto de matrices necesario y ejemplo particular de su implementación para la arquitectura de este proyecto. En el caso [A], debido a que la cámara virtual azul es hija del portal rojo, basta con pre-multiplicar la matriz de modelado de la cámara principal por la matriz de modelado inversa del portal azul, para obtener la matriz correcta en el sistema de coordenadas del portal rojo. En el caso [B], simplemente se necesita una transformación adicional para pasar del sistema de coordenadas local del portal rojo al global de la escena, esto se consigue pre-multiplicando el producto anterior por la matriz de modelado del portal rojo. En este proyecto se opta por mantener las cámaras virtuales como hijas del portal contrario, luego se usa el caso [A].

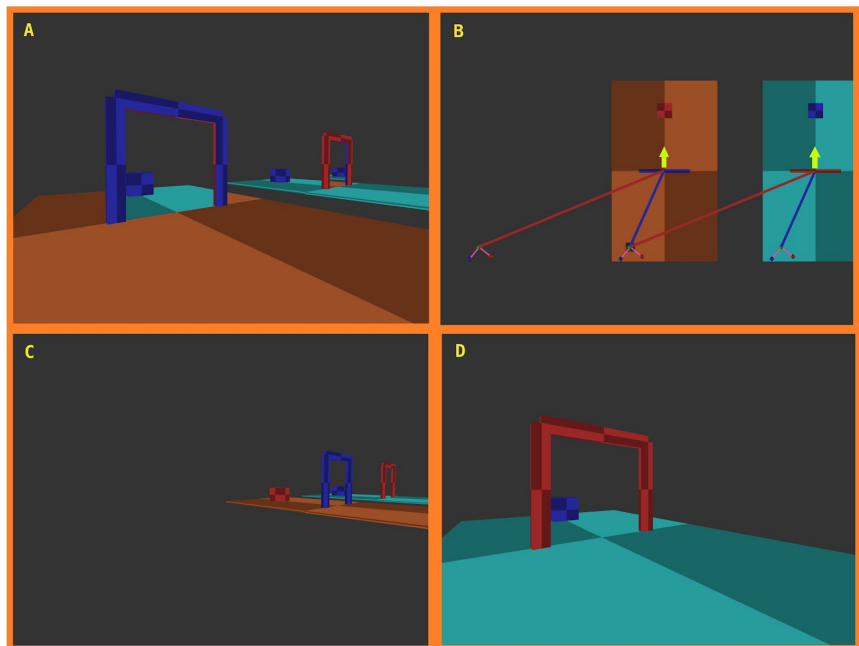
Ya que hay dos portales, que incluso pueden aparecer en pantalla simultáneamente, se necesitan dos cámaras virtuales para poder observar la escena desde el punto de vista principal y desde el punto de vista relativo a cada portal. Aunque un portal no se vea, su cámara virtual permanece en la posición relativa correcta, ya que su función real es la representación en la escena de su transformación completa (matriz de modelado). Tanto en una implementación de portales basada en texturas como en una que utilice *stencil-buffer*, es necesario calcular y mantener las posiciones de estas cámaras virtuales.

A continuación se muestran una serie de figuras para demostrar diferentes escenas con diferentes transformaciones de ambos portales y jugador. Todas ellas se componen de cuatro vistas:

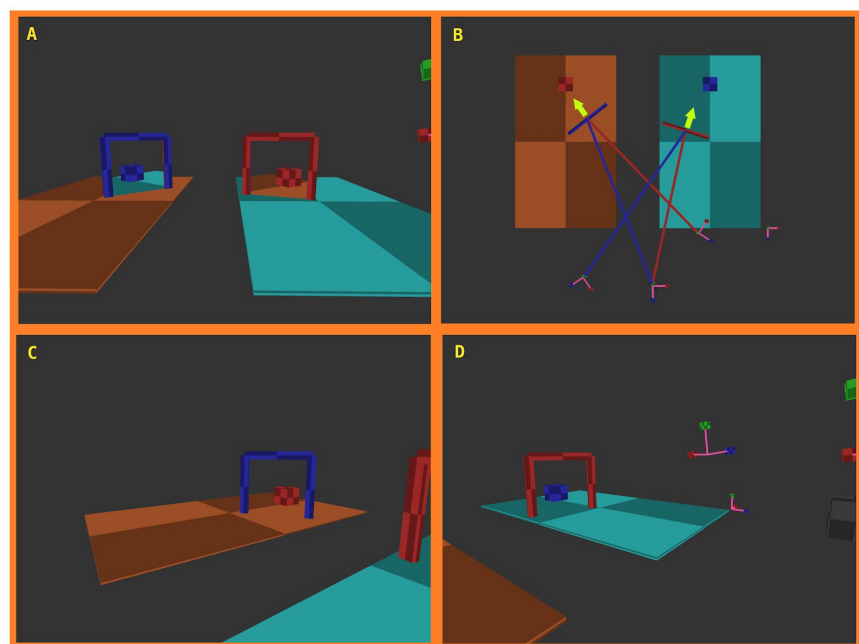
- En la vista [A] se muestra la escena en primera persona, en ella se ven los portales y a través de ellos correctamente.
- En la vista [B] se muestra la escena en cenital, y en ella se marcan dos líneas *rojas* que conectan [jugador → portal rojo] y [cámara virtual roja → portal azul]; estas líneas ayudan a visualizar cómo ambos se encuentran en la misma posición local respecto a los portales. También se trazan dos líneas azules para la otra cámara virtual.
- En [C] y [D] se muestra la escena en primera persona (portales desactivados) a través de los puntos de vista de las cámaras virtuales roja y azul, respectivamente.



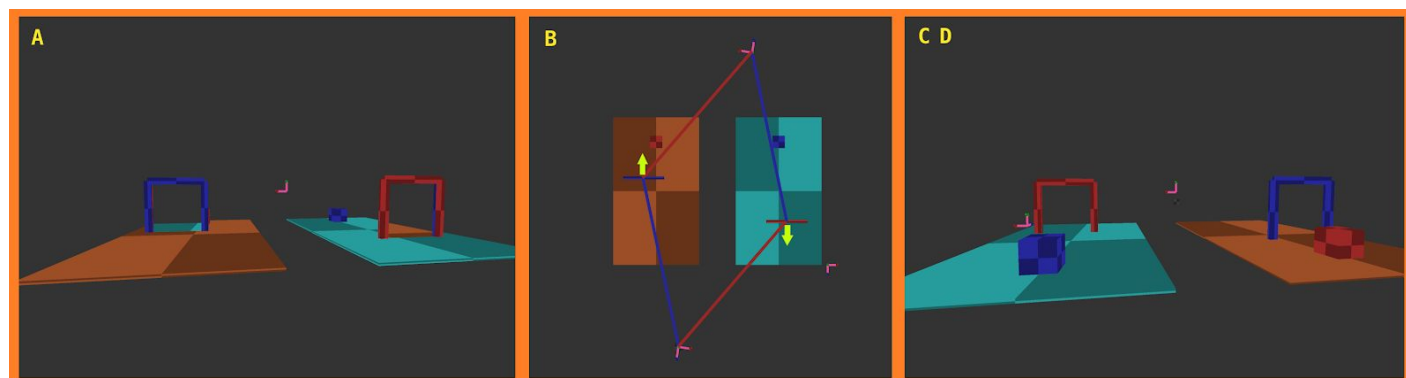
[Figura 6.1.4]: *escena sencilla, sin rotaciones. Simplemente es la primera vez que se pueden observar ambos portales simultáneamente. De nuevo, las vistas de las cámaras virtuales coinciden de forma exacta con el contenido que se ve a través de los portales en la vista principal.*



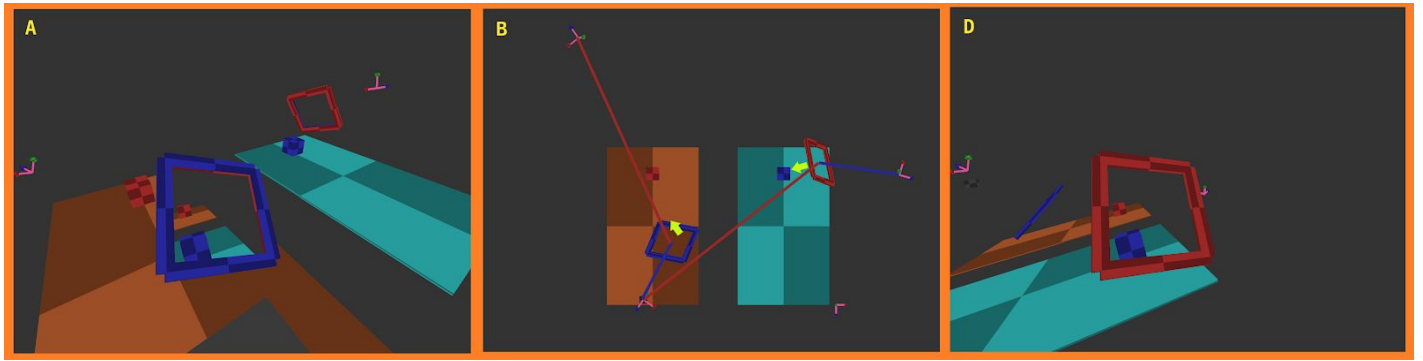
[Figura 6.1.5]: otra escena sencilla sin rotaciones. Pero en este caso es curioso como se puede observar el cubo azul tres veces.



[Figura 6.1.6]: primera escena con rotaciones. Ambos portales han sido ligeramente rotados.



[Figura 6.1.7]: escena bastante interesante. Debido a que uno de los portales ha sido girado 180 grados, se produce una simetría que hace que ambas cámaras virtuales se superpongan. Debido al giro, a través de los portales sólo se ven las plataformas, no cubos.

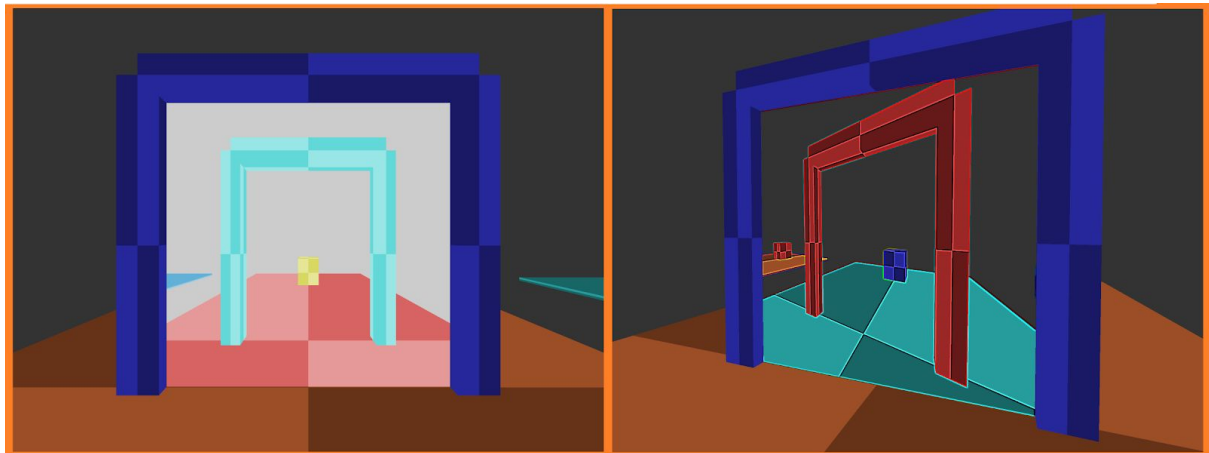


[Figura 6.1.8]: última escena. Ejemplo de rotaciones y traslaciones de ambos portales sobre todos los ejes (en las anteriores sólo se rota alrededor del eje Y). En este caso las líneas de la vista superior [B], se trazan hasta los centros de los portales (no activados en esta vista). Sólo se muestra la vista [C] desde la cámara virtual azul, ya que la otra apunta hacia el vacío (dirección opuesta a la cara del portal azul observado en la vista [A]).

Ya sólo falta explicar el uso concreto de la API para el método de implementación escogido, renderizado mediante texturas enlazadas a un *framebuffer*. Esto significa que los pasos siguientes difieren en caso de utilizar el método con *stencil-buffer*. El proceso de dibujado de la escena completa, que tiene lugar cada vuelta de bucle, ha de realizar los siguientes pasos generales en orden para dibujar los portales correctamente (el orden de portales no importa):

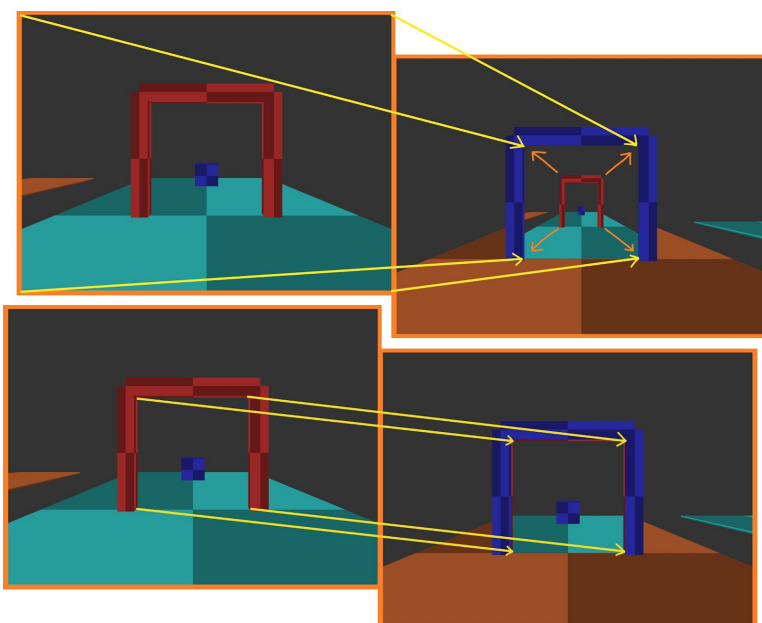
1. Desactivar la superficie del portal rojo (para que no bloquee la vista).
2. Activar el punto de vista de la cámara virtual azul.
3. Activar el *framebuffer* conectado a la textura del portal azul y borrar su contenido.
4. Renderizar la escena completa (que se escribe en la textura del portal azul).
5. Desactivar la superficie del portal azul (y reactivar la anterior, del portal rojo).
6. Activar el punto de vista de la cámara virtual roja y borrar su contenido.
7. Activar el *framebuffer* conectado a la textura del portal rojo y borrar su contenido.
8. Renderizar la escena completa (que se escribe en la textura del portal rojo).
9. Activar la superficie de ambos portales (ya tienen sus texturas correctas).
10. Activar el punto de vista de la cámara principal.
11. Activar el *framebuffer* por defecto, conectado al *back-buffer* de la pantalla y borrar su contenido. Respecto a todos estos *framebuffer*, también se borran sus *depth-buffers*.
12. Finalmente renderizar la escena completa.

A este proceso se pueden añadir multitud de detalles. Por ejemplo, en la implementación de este proyecto cada textura de portal se dibuja previamente en un *framebuffer* auxiliar, esto permite poder aplicar cualquier efecto de post-procesado. Antes de dar por terminado la implementación de la “vista a través de la superficie del portal”, es interesante comentar un error muy frecuente que puede ocurrir al asignar la textura de un portal a su superficie. Concretamente, este fallo consiste en asignar la textura *completa* de la vista de la escena desde una cámara virtual a su portal.



[Figura 6.1.9]: *Imágenes en primera persona en las que se produce este fallo. Se ha añadido una muestra de los efectos de post-procesado recientemente comentados (izquierda negativo y derecha contraste aumentado). Este fallo es mucho más notable con la distancia y a la mínima rotación (como se observa en la imagen derecha).*

La solución a este problema se basa en asignar las coordenadas en pantalla de los vértices del portal, como coordenadas de textura a utilizar en la imagen de la escena completa. Esto tiene el efecto de *recortar* la imagen y consecuentemente descartar todo el contenido que queda fuera de la superficie del portal. Este envío de datos se suele implementar en los *Shaders*.



[Figura 6.1.10]: *Esquema sencillo que explica el fallo (textura asignada completamente), y su solución (textura asignada parcialmente). A través del portal se tiene que ver la vista de la cámara virtual, y de esta vista, exclusivamente sólo la parte que se observa dentro del portal.*

```

//VERTEX SHADER
//just normalize xy coordinates from [-w, w] to [0, w]
TexCoordinates = (gl_Position.xy + gl_Position.w) / 2.0;
w = gl_Position.w;

//do not normalize to [0, 1] screen per vertex
//this could only work with polygons facing the camera
//w needs to be interpolated for each fragment
//send w as OUT to the fragment shader
//OUT variables get interpolated

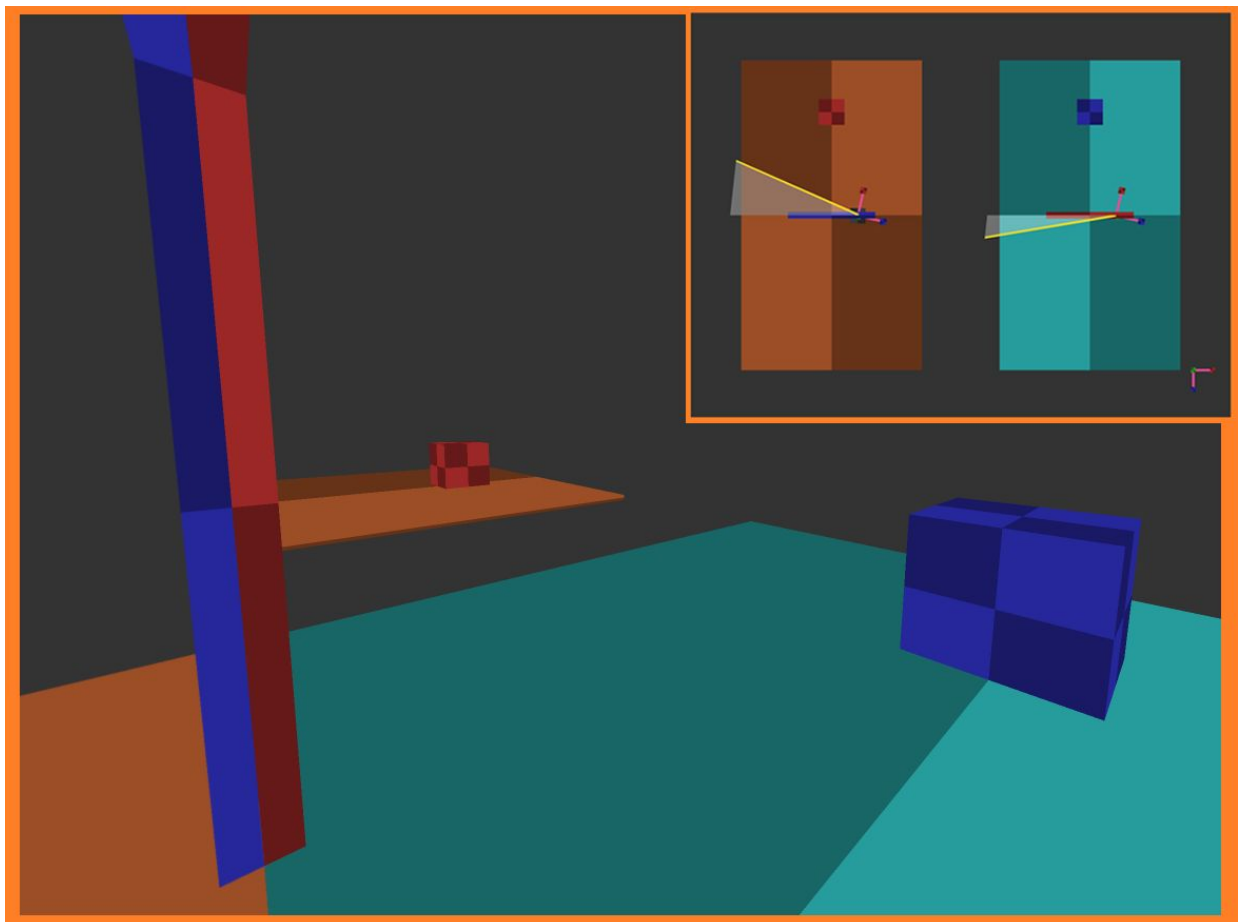
//FRAGMENT SHADER
//now normalize by w correctly interpolated
TexCoordinates /= w;

//usual texturing using uv coordinates
FragColor = texture(texture0, TexCoordinates);
//posibly to add coloring
FragColor *= vec4(color, 1.0);

```

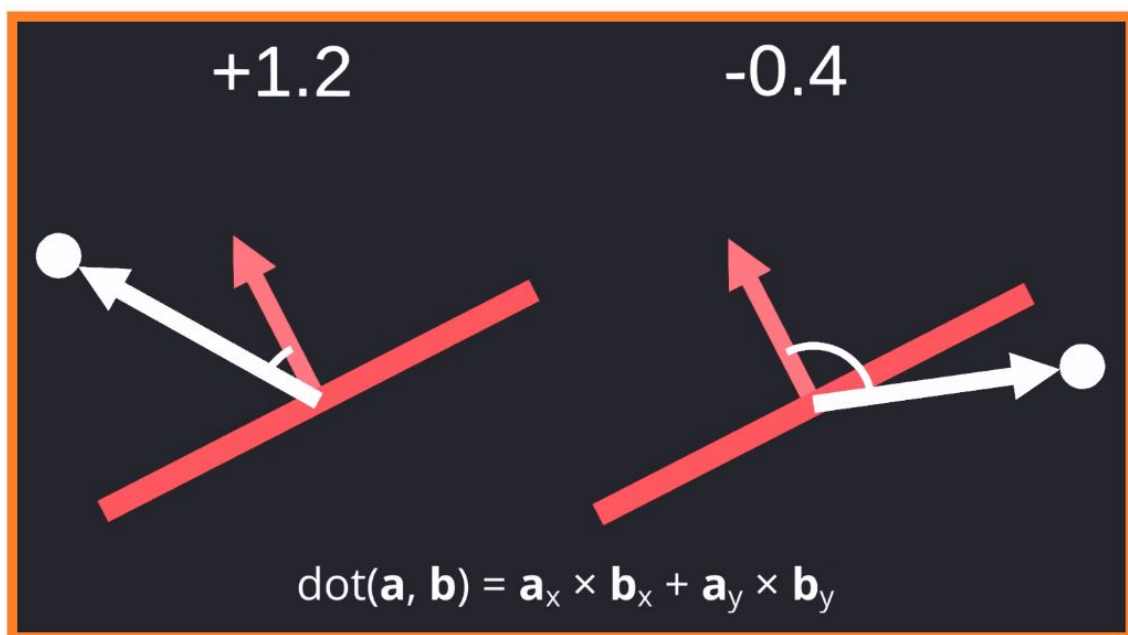
[Figura 6.1.11]: Muestra de las fórmulas utilizadas en esta implementación para transformar las coordenadas de textura de los portales, se emplean desde los shaders y permiten asignar correctamente la textura a sus superficies.

6.2. Desplazamiento en primera persona



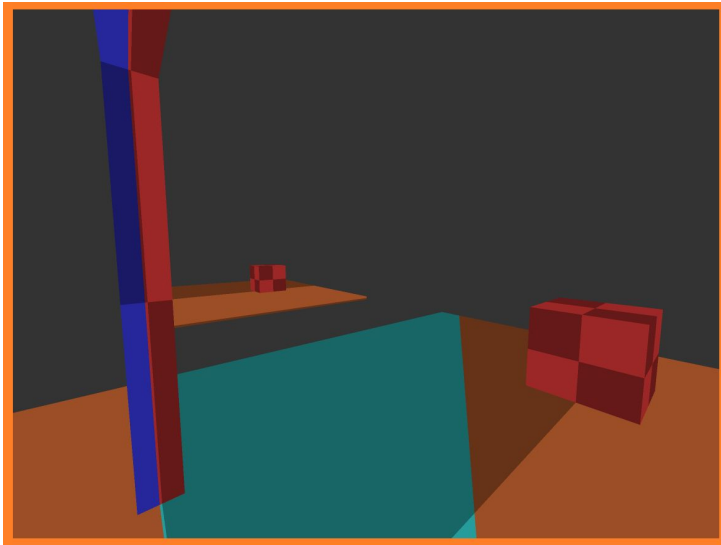
[Figura 6.2.1]: Vista en primera persona (en perspectiva), en la que se atraviesa satisfactoriamente la superficie del portal. El jugador se ha posicionado justo en su límite, por ello se pueden ver ambos lados a la vez (cada uno desde un portal diferente).

Esta parte es relativamente corta, pues como ya fue indicado, la simulación física no se prioriza y se implementa de forma simplificada. La lógica se reduce a comprobar en cada vuelta del bucle si el jugador ha cambiado de lado respecto a alguno de los portales. Esto se consigue almacenando la información de posición del ciclo anterior y realizando una comparación con la actualizada. En caso de cambio de lado, se desplaza el jugador a la misma posición local relativa pero del portal contrario. Estas comparaciones se basan en el producto escalar entre el vector normal de la superficie del portal (cara frontal), y el vector de posición del jugador en coordenadas locales del mismo portal; el resultado de esta operación varía de signo según el lado. Por optimización, en realidad la información que se guarda cada vuelta del bucle es sólo el signo obtenido del producto escalar, y además sólo se realizan comparaciones dentro de una esfera de interés de radio aproximado [*Anchura del portal* * 0.6 + *Anchura del jugador* * 0.6] (se almacena el cuadrado y se compara directamente con la longitud cuadrada del vector posición del jugador respecto al portal). Se ha de tener cuidado con los sistemas de física asíncronos (diferente frecuencia de actualización), ya que es necesario que esta comprobación tenga lugar en cada ciclo para no haya desincronización con el renderizado.

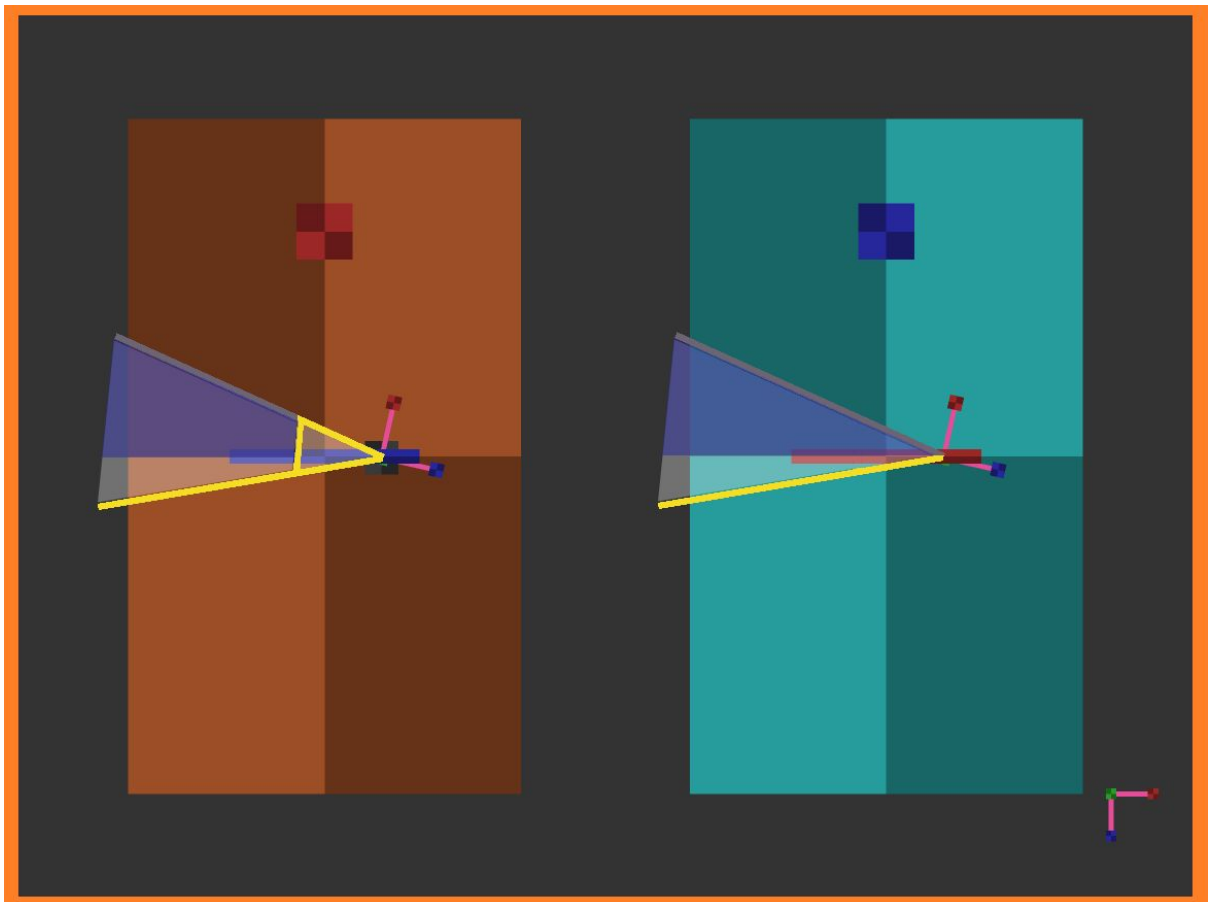


[Figura 6.2.2]: Recurso editado de Sebastian Lague, Coding Adventures [68]. Producto escalar ilustrado. Cuando la posición local del jugador se encuentra en el lado que coincide con la dirección del portal (vector normal, cara frontal), el resultado es positivo. Si por el contrario, está situado al otro lado, el resultado es negativo. Si ambos vectores son perpendiculares el resultado es cero, pero esto no es problema para la implementación.

Debido al uso de texturas, surge un problema con la superficie de los portales. El plano cercano de la cámara en perspectiva puede cortarlo al acercarse, y todo lo que se encuentra entre este plano cercano y la cámara se descarta. Esto produce un artefacto siempre que se atraviesa el portal. En el método con *stencil-buffer* esto normalmente sólo ocurre en caso de que el portal sea casi coplanar con la superficie de una pared, entonces el plano cercano de la cámara corta igualmente la pared.



[Figura 6.2.3]: *Misma vista en primera persona que en la figura [6.2.1]. En este caso se deja ver el artefacto gráfico que se produce al atravesar el portal. En la siguiente figura se explica su naturaleza.*



[Figura 6.2.4]: *Vista cenital de las escenas anteriores [6.2.1] y [6.2.3], en la imagen se ha añadido la representación (no a escala) de los espacios de vista de las cámaras. A la izquierda se puede observar la cámara principal y se destaca su plano cercano (única línea amarilla casi vertical). Por delante del plano cercano todo funciona correctamente como en situaciones anteriores: la porción de la escena que se encuentra al otro lado del portal, se sustituye por la vista de la cámara virtual. Pero la zona entre el plano cercano y la cámara principal falla, se descarta la superficie del portal y simplemente se ve detrás del mismo.*

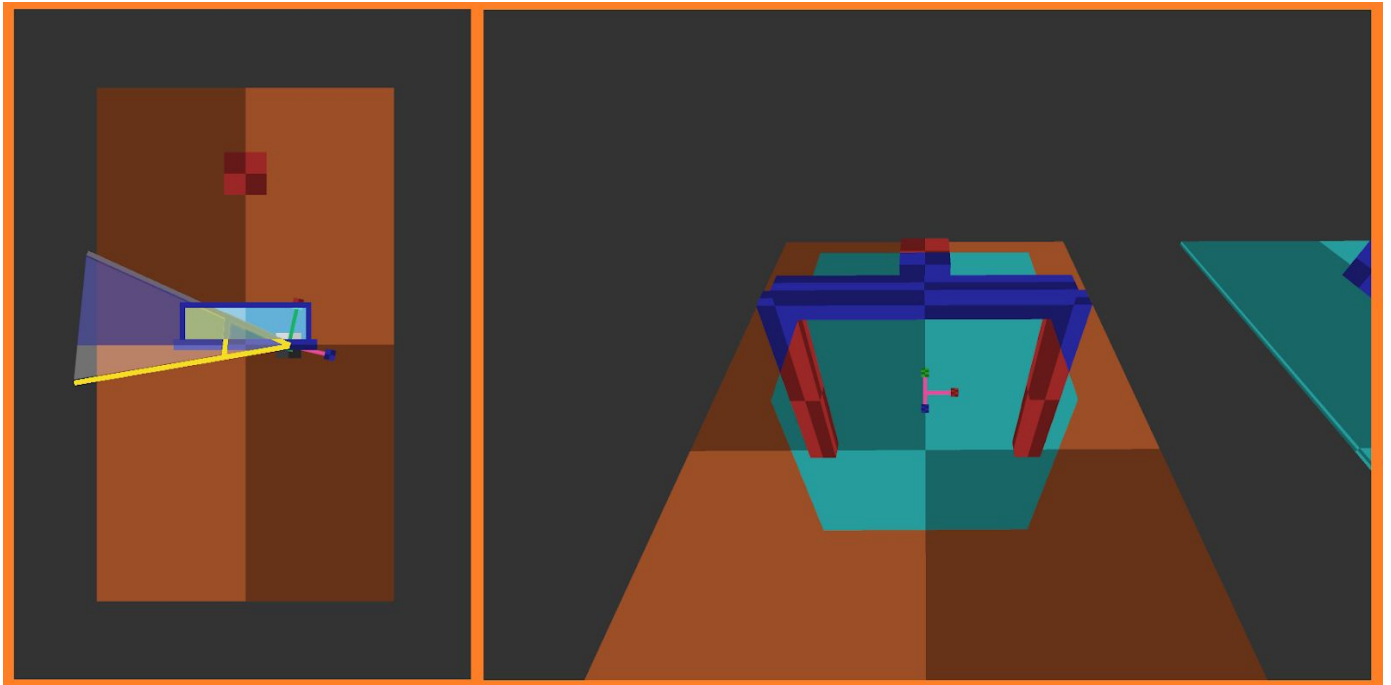
Existen múltiples formas de solucionar este problema, las más razonables dependen de la distancia del jugador al portal:

- Reducir la distancia del plano cercano de la cámara lo suficiente como para no cortar el portal, a medida que el jugador se acerque. Este método no es muy bueno, ya que se edita demasiado la matriz de proyección y además no se asegura de que no se produzca ningún artefacto.
- Aumentar el grosor de la superficie del portal, es decir, que en vez de ser plana sea un prisma muy fino. Además, las caras internas de este prisma deben ser visibles (es posible que por defecto no lo sean), esto en inglés se conoce como “*culling*”. Entonces basta con ajustar la anchura del prisma para que el plano cercano de la cámara siempre quede en su interior. Desde el interior no hay problema, porque la textura se asigna correctamente respecto a sus coordenadas en pantalla.

En la implementación concreta, en cada ciclo y si el jugador ha entrado dentro de la esfera de interés de un portal (o lo ha atravesado), se utiliza la información referente al lado en el que se encuentra, para expandir la superficie (generar el pequeño prisma) hacia el lado contrario. De esta forma la superficie original queda intacta debido a que es coplanar con la cara exterior del nuevo prisma; la transición es inapreciable. Por optimización no se ajusta el tamaño de la extrusión dinámicamente al límite exacto del plano cercano de la cámara, sino que se supone el peor caso: se expande una única vez, hasta la distancia máxima entre la cámara y el plano cercano.



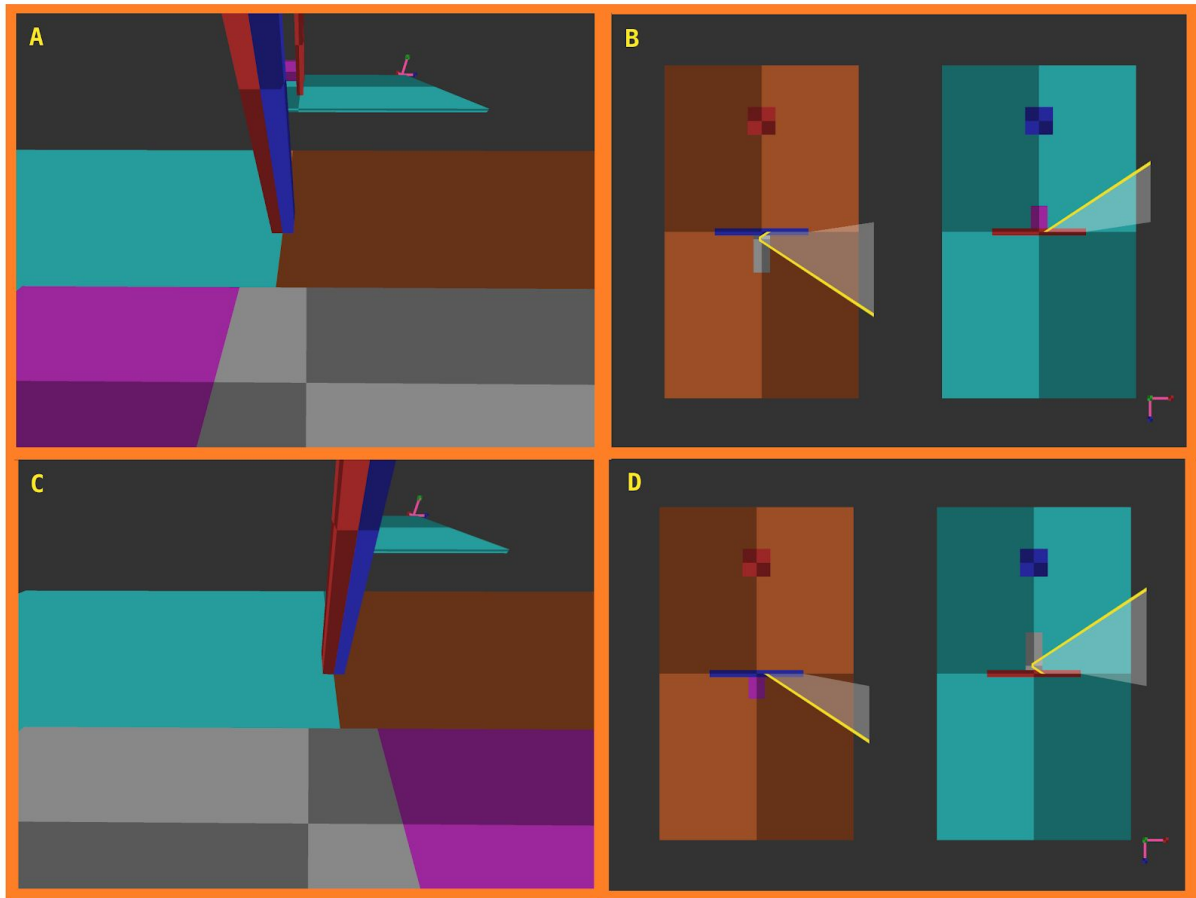
[Figura 6.2.5]: Sencilla muestra de las caras del interior del cubo rojo. Lo común es que estas caras sean invisibles ya que normalmente sólo se observan las exteriores. En este caso las superficies de los portales necesitan que sus caras interiores sean visibles.



[Figura 6.2.6]: A la izquierda, se muestra de nuevo la vista cenital de la escena anterior. En esta ocasión se ha añadido una representación exagerada de la expansión de la superficie del portal (rectángulo azul con colores invertidos en su interior). Ahora el plano cercano de la cámara queda dentro del prisma y por ello no se produce ningún artefacto. A la derecha, se muestra una vista en primera persona del prisma del portal (muy exagerado); como se puede ver, incluso en los laterales, la porción de la superficie del portal se asigna perfectamente las texturas de la vista de la cámara virtual. En el medio del portal se puede apreciar un eje de coordenadas, (disponibles para cualquier objeto de la arquitectura), que representa su origen y permite conocer su dirección.

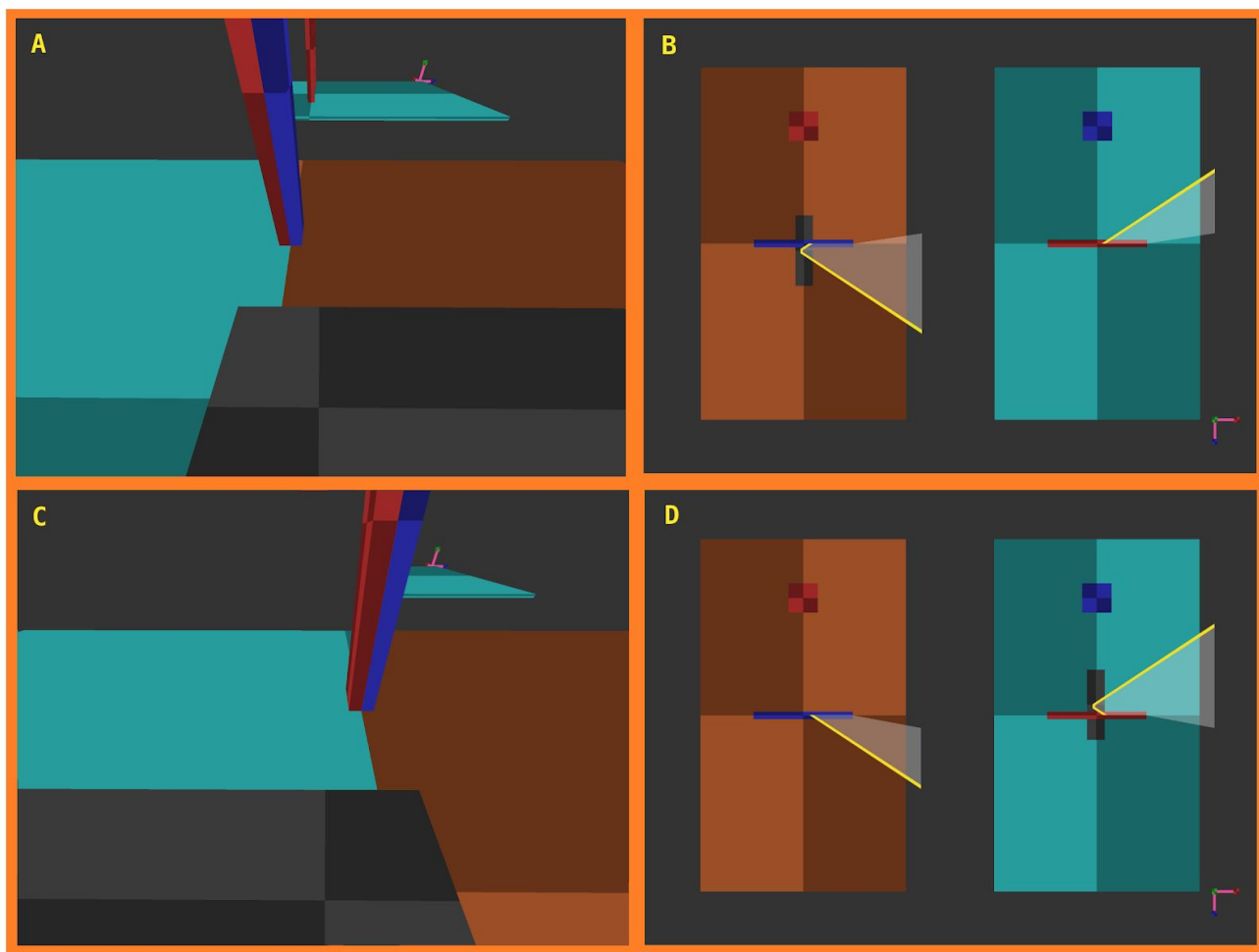
Actualmente, el cuerpo del jugador (pequeño cubo oscuro) se ve cortado parcialmente cuando se atraviesa un portal. Esto de momento es indiferente, ya que en primera persona los *cuerpos* (brazos de personaje, armas, etc) suelen renderizarse a posteriori, ignorando el *depth-buffer*. En el siguiente apartado, que trata de la vista externa de cómo un objeto traspasa un portal, el comportamiento correcto del cuerpo del jugador sí es parte de los objetivos; tanto en primera persona como desde cualquier punto de vista de la escena.

6.3. Vista externa de un desplazamiento



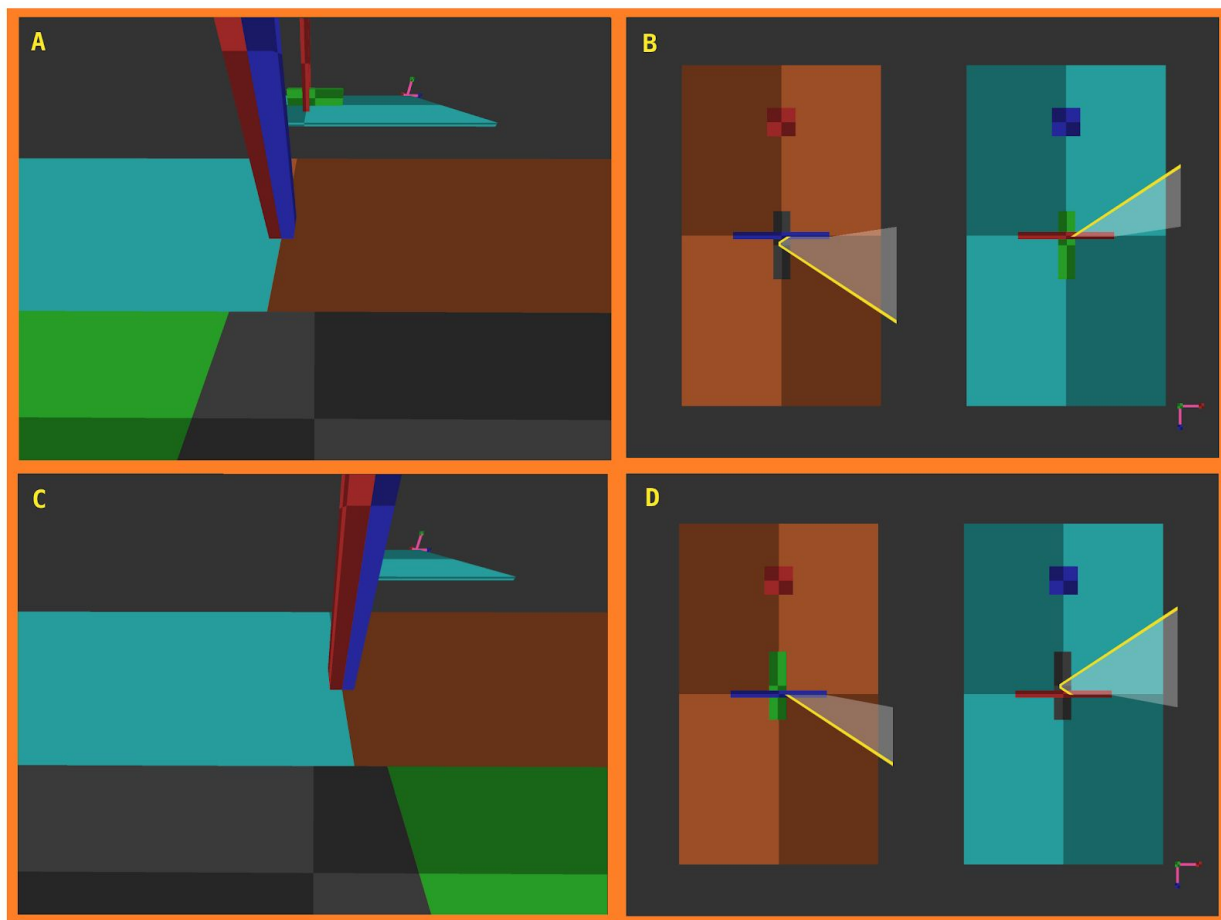
[Figura 6.3.1]: Esta figura muestra cómo se debe ver un objeto desde un punto de vista externo [B / D], al atravesar un portal, partido en dos y una fracción a cada lado. Se muestran distintas vistas del jugador en primera y tercera persona, el cuerpo ha sido alargado para que el efecto de corte sea más obvio. Esta versión final es muy interactiva: el jugador se vuelve de color claro al entrar en la esfera de interés del portal y el corte que sale por el otro lado es rosa. Como se puede apreciar en primera persona [A / C] no hay ningún error, encaja perfectamente. En [A / B], el objeto aún no ha cruzado el portal y poco a poco avanza hasta que su centro lo supera y se llega a [C / D], en estos los colores se han intercambiado debido a que ahora el original se encuentra al lado del otro portal. Es una transición continua sin desplazamientos bruscos apreciables. Este procedimiento es común para ambos métodos de implementación discutidos, texturas o stencil-buffer.

Para alcanzar el objetivo de este apartado, es decir el resultado mostrado en la figura [6.3.1], se requiere bastante trabajo. Por ejemplo, de momento, el desplazamiento es instantáneo y brusco, sin sensación de continuidad: en cuanto el centro del objeto (su *origen*) cruza el umbral del portal, éste se transporta de golpe, osea desaparece de un lado del portal y aparece al lado contrario del otro portal.



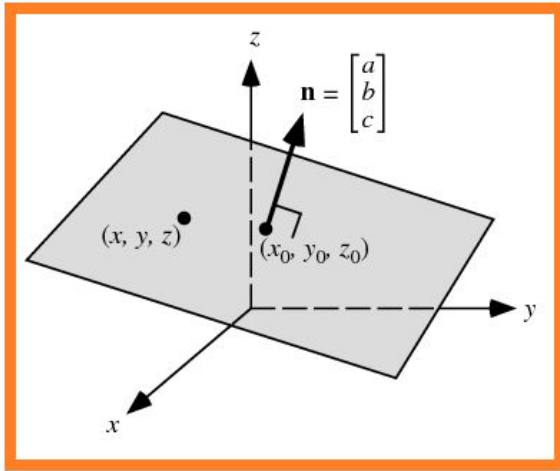
[Figura 6.3.2]: Esta figura muestra el comienzo de la implementación, lejano al éxito. Desde los puntos de vista externos [B / D], el objeto sobresale por ambos lados del portal y se desplaza de golpe sin continuidad. En los puntos de vista en primera persona [A / C], el objeto es cortado y ocluido por el portal. En esta versión la proximidad al portal no interactúa con el color del jugador.

Para obtener el corte del cuerpo del jugador y poder posicionarlo junto al portal contrario, se utiliza una copia del cuerpo entero; esto es mejor que crear o editar geometría dinámicamente. La copia es estática, se genera al principio de la ejecución, sin embargo, si se habilita la modificación del cuerpo del jugador (por ejemplo, estirarlo como en las figuras previas), estos cambios también deben ser reflejados en la copia. Cuando el jugador entra en la esfera de interés de un portal, se coloca su copia en la misma posición local, pero respecto al portal contrario. En el evento contrario, cuando el jugador sale de una zona de interés, se vuelve a ocultar su copia. Por ahora no se hace ningún corte, cuando ambos objetos son visibles, lo son al completo. Aun así se ha conseguido algo relevante, ahora existe continuidad en el movimiento, ya no se producen desplazamientos bruscos del jugador.



[Figura 6.3.3]: En esta figura se muestran varios puntos de vista en los que la copia del jugador se encuentra activada, esto se debe a que se encuentra dentro de la esfera de interés. El cuerpo copiado se representa con color verde, el color del jugador aún no se altera por estar dentro de la zona. Desde los puntos de vista externos [B / D], se pueden ver ambos objetos alineados y sobresaliendo por ambos lados; nuevamente, esto no es un resultado aceptable, pero por ahora se ha logrado reproducir un movimiento continuo. En los puntos de vista en primera persona [A / C], el jugador sigue sufriendo cortes indeseados debidos al prisma que forma la superficie del portal (en [C] se puede observar claramente la cara exterior). Pero su apariencia ha mejorado considerablemente, al estar alineados, la copia verde que se ve a través del portal parece una extensión del cuerpo del jugador.

Es el momento de añadir *dos* planos de corte para delimitar la geometría que debe renderizarse y la que no. Internamente funcionan utilizando el producto escalar, operación ya usada previamente para determinar en qué lado de cada portal se encuentra el jugador. En esta situación, los planos de corte siempre serán coplanares a las superficies de los portales. Precisamente, para cortar correctamente el cuerpo, un plano se sitúa sobre el portal de entrada del jugador, y su normal apunta hacia el lado que se quiere vacío (el de la cara contraria a la atravesada). El segundo plano de corte, se dispone sobre el otro portal y su normal apunta en el sentido contrario, esto permite cortar el lado contrario de la copia.



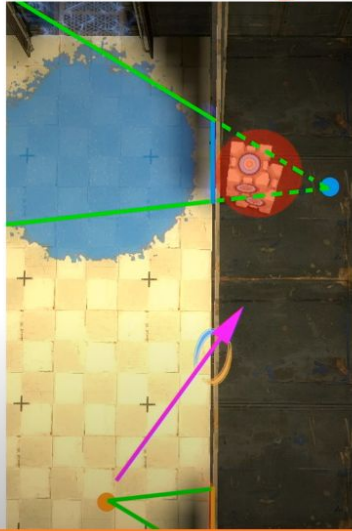
[Figura 6.3.4]: Esquema de un plano definido por una normal y un punto [69]. En OpenGL se representa con un vector de 4 componentes, compuesto por dicho vector normal $[N]$ y el producto escalar entre $[-N]$ y un punto dado $[P]$.

$$[Nx, Ny, Nz, -N \cdot P]$$

Estos planos de corte (su representación en forma de vector 4D), se calculan y se activan cuando el jugador entra en la esfera de interés de un portal, y otra vez cuando lo atraviesa. Se debe indicar a *OpenGL* que se activa o desactiva (*importante*) el plano de corte personalizado, además se debe enviar al *vertex shader* su vector 4D como variable *uniform*. Finalmente dentro de este *shader*, se calcula el producto escalar entre la posición global del objeto en cuestión (el jugador o su copia), y el plano de corte enviado; el resultado se almacena en una variable de salida del *shader* que indica a *OpenGL* si debe descartar el fragmento o no; sólo descarta si el plano de corte personalizado se encuentra activado.

A estas alturas ya se ha obtenido la implementación completa mostrada en la figura [6.3.1]. Así como este proceso (uso de modelos duplicados y planos de corte), es común entre los dos métodos de implementación discutidos (texturas o *stencil-buffer*), también existe un problema latente que afecta a ambos. Tiene que ver con la posibilidad de que algún objeto de la escena se interponga entre una cámara virtual y el portal por el que necesita ver a través. Esto efectivamente es un error que ha permanecido oculto desde el principio, desde el primer apartado donde se calcula las posiciones relativas de las cámaras virtuales. No ha sido relevante ya que en la escena inicial, apenas hay objetos y además la prioridad es el funcionamiento en primera persona, osea alrededor del jugador. De hecho el error ocurre en primera persona, sólo si se abusa de alguna posición y perspectiva que coloque uno de los cubos en el lugar preciso; pero este error es muy específico y complicado de entender, no es constructivo ni comentarlo tan temprano. Ahora es un buen momento, debido a que sus posibles soluciones están todas relacionadas con el uso de planos de corte.

Rendering: Banana Juice

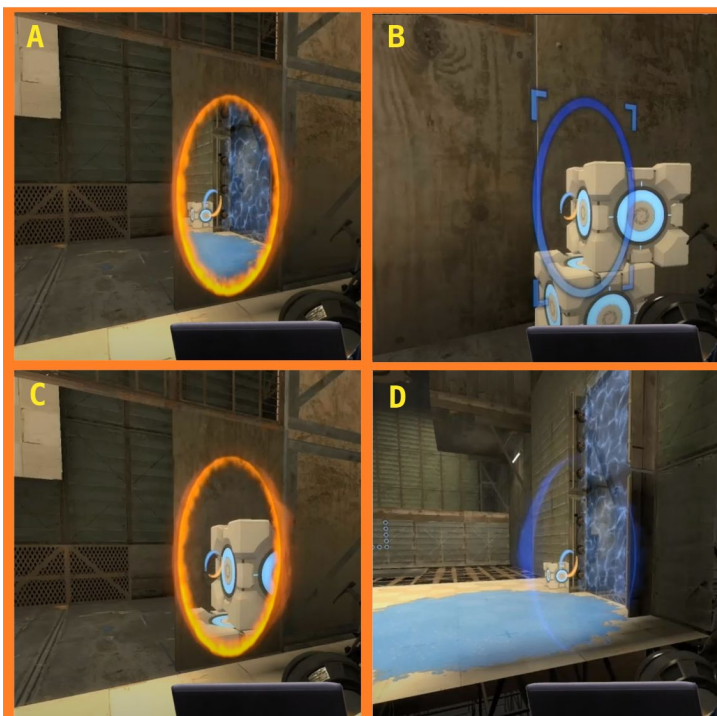


Shorthand for a complicated problem while trying to make it obvious that explanation was required.

When rendering a portal view. Objects between the and the exit portal can occlude the view.



[Figura 6.3.5]: Captura de pantalla de parte de la conferencia dada por Dave Kircher para CS50 [63]. Retomando el caso de estudio, durante el desarrollo de Portal el equipo se topa con un problema inesperado y difícil de explicar. Literalmente lo denominan “zumo de plátano”, para que sea obvio que es un problema complicado que necesita explicación. En la parte izquierda de la imagen, se puede ver un esquema con los espacios de vista de las cámaras, la inferior (punto naranja) es la cámara principal y parte de su espacio de vista se proyecta y sale por la superficie del otro portal. Este espacio de vista que sale del portal, no es otro que el de la cámara virtual del portal asociado (punto azul); esta cámara se encuentra en la posición correcta, que resulta ser tras la pared. Entre el portal azul y la cámara hay una montaña de cubos, y estos son los causantes del problema ya que ocuyen la vista de la cámara virtual.

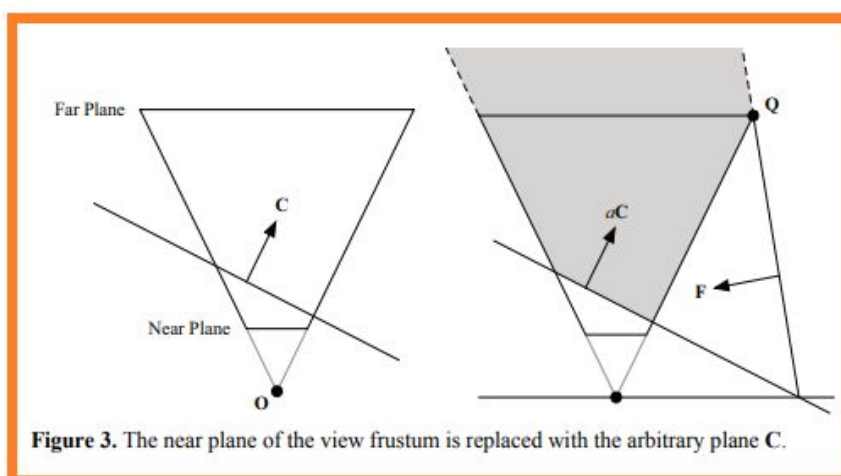


[Figura 6.3.6]: Recurso editado de la conferencia dada por Dave Kircher [69]. Vista en primera persona del problema en cuestión. En [A] se observa la vista correcta desde la cámara principal, pero en [B] se muestra por qué falla. Como la vista de la cámara virtual se encuentra oculta por los cubos y la pared, no se puede ver al otro lado de la superficie del portal. Esto provoca [C], donde los objetos que estorban se muestran a través del portal, en vez de la escena esperada. En [D] se habilita un plano de corte para evitar que la cámara sea oculta y poder obtener [A].

Estos planos de corte son algo más interesantes que los anteriores, los cuales sólo se utilizaban para cortar el cuerpo del jugador o su copia; en el renderizado de todas las vistas. En cambio, estos planos se activan durante todo el renderizado de la escena desde el punto de vista de cada cámara virtual, y se descartan fragmentos de cualquier objeto que se sitúe entre el portal y la cámara. Estos planos, también son coplanares con las superficies de los portales. Entre las formas razonables de implementarlos se encuentran las dos siguientes:

- Reutilizar los mismos planos de corte que se usan para el jugador y su copia, mantenerlos activos durante los pases de dibujo de las cámaras digitales.
- Utilizar una matriz de proyección especial que transforme el plano cercano de la cámara para que sea oblicuo y siempre coincida con la superficie del portal. Consecuentemente, esto descarta todos los fragmentos entre el portal y la cámara, ya que su plano cercano es coplanar al del portal.

La primera opción es perfectamente válida y su implementación (*dada la actual*) no es muy complicada (por ejemplo, se usa en la saga de *Portal*). Pero la segunda opción, es muy interesante y didáctica, por ello se va a desarrollar y explicar a continuación. Con esta técnica, esencialmente se crea un plano de corte personalizado a partir de una matriz de proyección en perspectiva. Esto tiene como ventaja no necesitar modificaciones en los shaders existentes, además de funcionar directamente en tarjetas gráficas antiguas que carezcan de soporte nativo para planos de corte adicionales. Tiene como consecuencia negativa, que el plano lejano se desplaza al infinito y esto provoca repercusiones en la precisión del *depth-buffer* (casi inapreciables, si se usa un *depth-buffer* de precisión estándar).



[Figura 6.3.7]: Eric Lengyel [70]. En la izquierda, esquema que muestra el espacio de vista de una cámara en perspectiva. Se quiere reemplazar el plano cercano por el plano arbitrario [C]. En la derecha, tras conseguirlo se pierde el plano lejano, se sombrea el nuevo espacio.

```

//http://terathon.com/code/oblique.html
//edited for glm matrix layout and opengl version
void SampleScene::modifyProjectionMatrixOptPers(
    glm::mat4 & proj, glm::vec4 const & clipPlane) {

    // Calculate the clip-space corner point opposite the
    // clipping plane: (fsgn(clipPlane.x),fsgn(clipPlane.y),1,1)
    // and transform it into camera space by multiplying it
    //by the inverse of the projection matrix

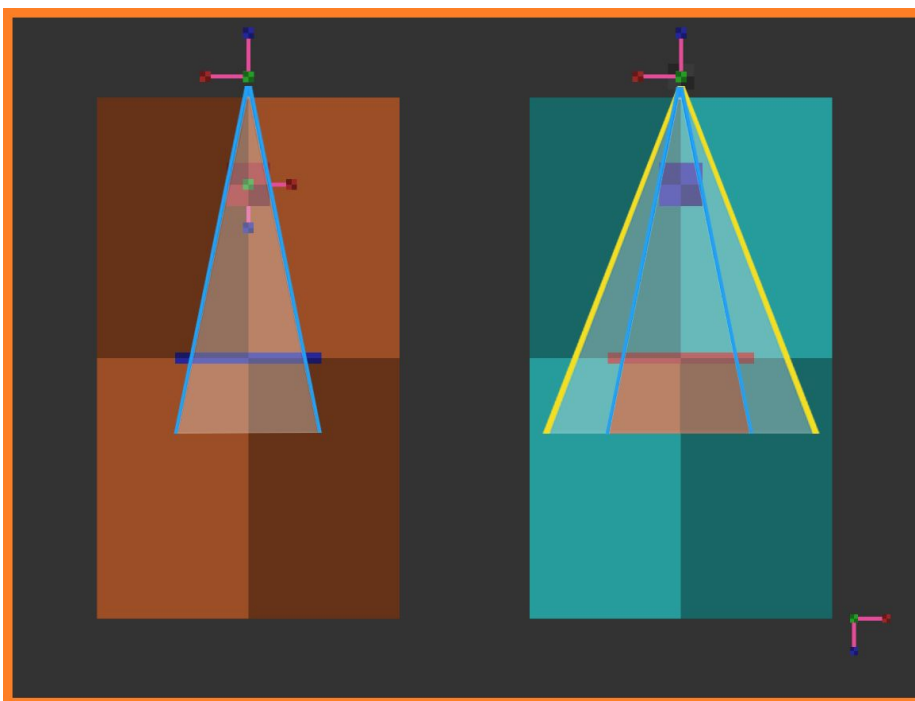
    glm::vec4 q;
    //optimized multiplying by inverse:
    q.x = (fsgn(clipPlane.x) + proj[2][0]) / proj[0][0];
    q.y = (fsgn(clipPlane.y) + proj[2][1]) / proj[1][1];
    q.z = -1.0F;
    q.w = (1.0F + proj[2][2]) / proj[3][2];

    // Calculate the scaled plane vector
    glm::vec4 c = clipPlane * (2.0F / glm::dot(clipPlane, q));

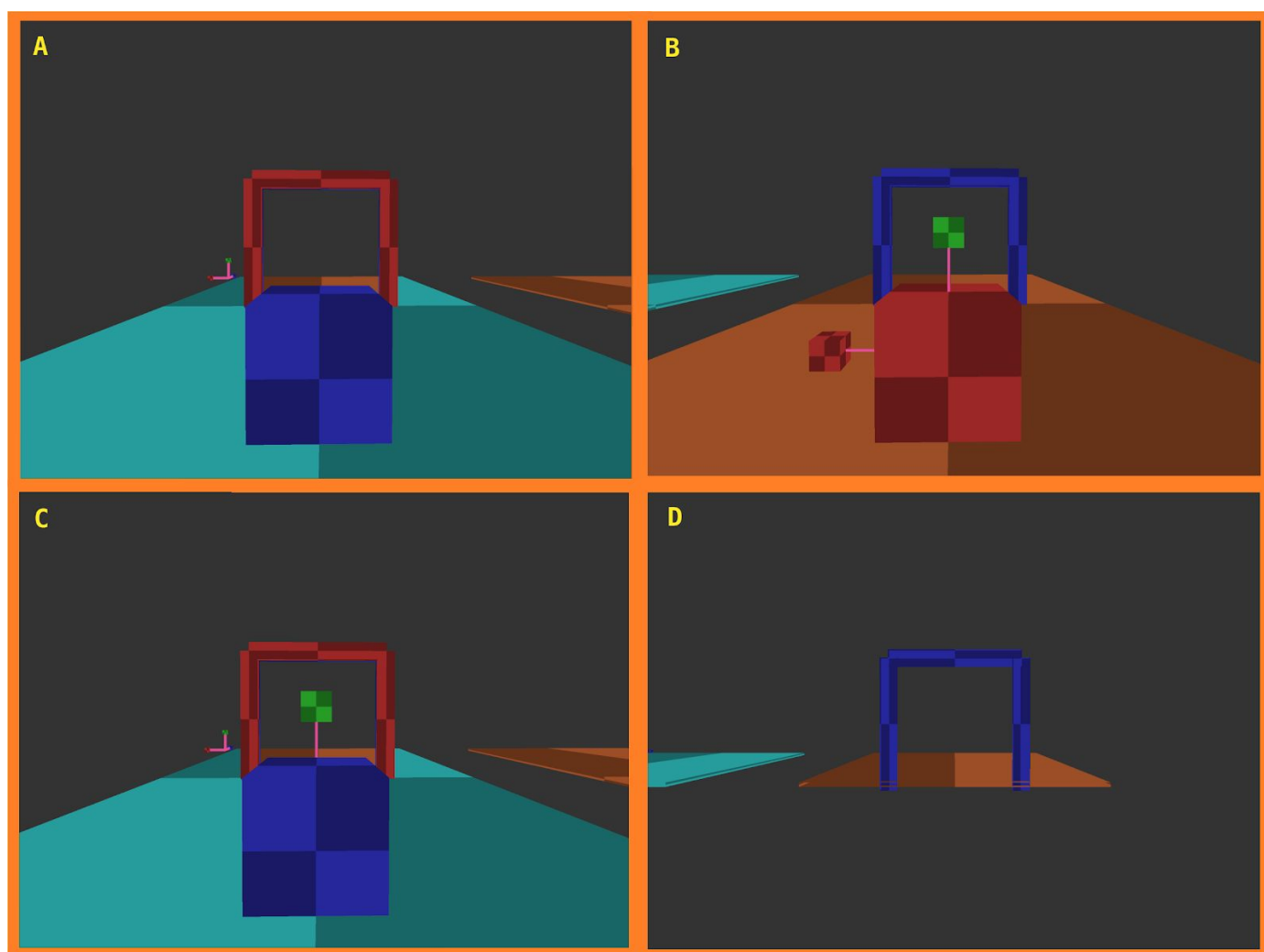
    // Replace the third row of the projection matrix
    proj[0][2] = c.x;
    proj[1][2] = c.y;
    proj[2][2] = c.z + 1.0F;
    proj[3][2] = c.w;
}

```

[Figura 6.3.8]: Eric Lengyel [70]. Muestra de la implementación y sus fórmulas. Esta función recibe referencias a la matriz de proyección y a un plano arbitrario dentro de su espacio de vista. Se modifica la matriz para que dicho plano sea su plano cercano. Esta es la implementación que forma parte de la arquitectura del proyecto, está adaptada para GLM.

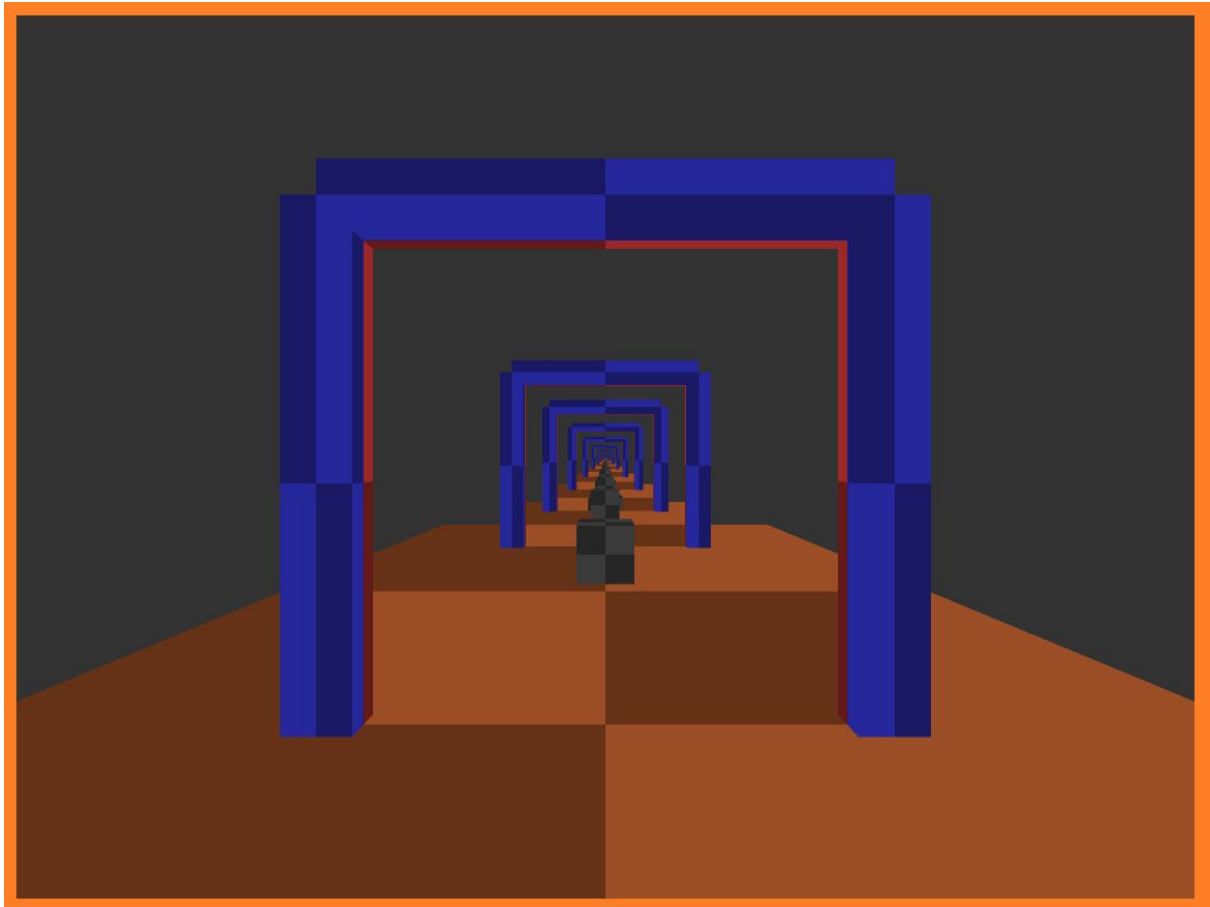


[Figura 6.3.9]: Esquema sobre la escena de pruebas en el que ocurre “banana juice”, al igual que en la figura [6.3.5]. La vista de la cámara virtual dispuesta a la izquierda, se encuentra parcialmente ocluida por los ejes de coordenadas del cubo rojo.



[Figura 6.3.10]: *Vistas en primera persona de la escena esquematizada en la figura anterior [6.3.9]. Se compara cómo afectan los planos de corte sobre los distintos puntos de vista de forma paralela a la figura [6.3.6], pero en este caso son creados mediante matrices de proyección oblicua. En [A] se observa la vista correcta desde la cámara principal, y en [B] se muestra la razón por la que puede fallar si no se usa un plano de corte en la vista de la cámara virtual. Concretamente, parte de esta vista se encuentra oculta por los ejes de coordenadas del cubo rojo, por ello se oculta parcialmente la escena al otro lado del portal. Esto provoca [C], donde en el punto de vista principal, parte de dichos ejes de coordenadas estorban ya que se muestran sobre la superficie del portal, tapando parte de la escena esperada. En [D] se muestra el punto de vista de la cámara virtual en el que se utiliza un plano de corte (generado mediante matriz de proyección oblicua), el efecto que consigue es descartar todo fragmento entre la cámara y la superficie del portal. Efectivamente se logra evitar que dicha cámara se oculte y obtener la imagen correcta observada en [A]. Recapitulando, en [B / D] se compara el efecto de un plano de corte (sobre la vista de la cámara virtual), [D] con y [B] sin; y en [A / C] se compara el resultado que produce (sobre la vista principal), [A] con y [C] sin. Tanto este problema como su solución son comunes para una implementación basada en texturas o stencil-buffer.*

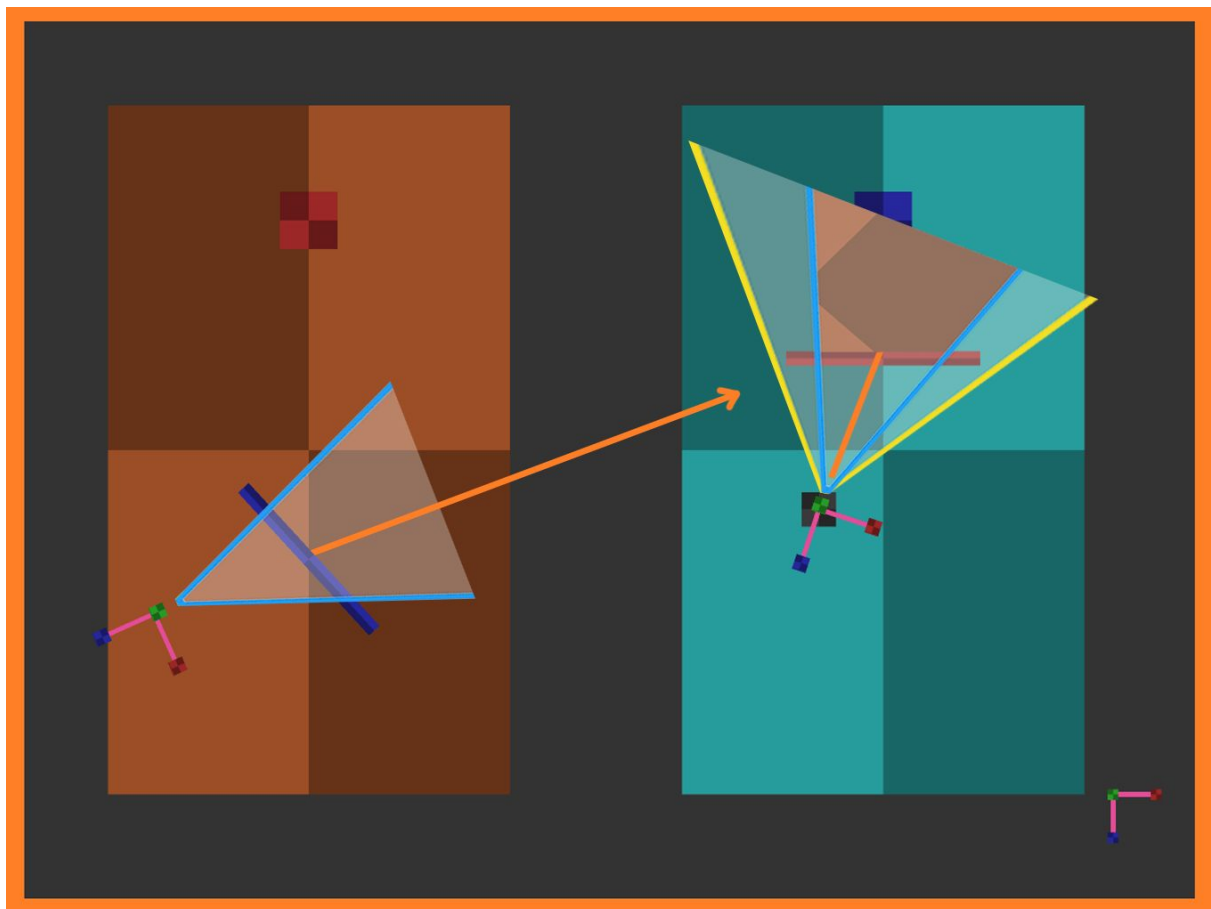
6.4. Vista recursiva



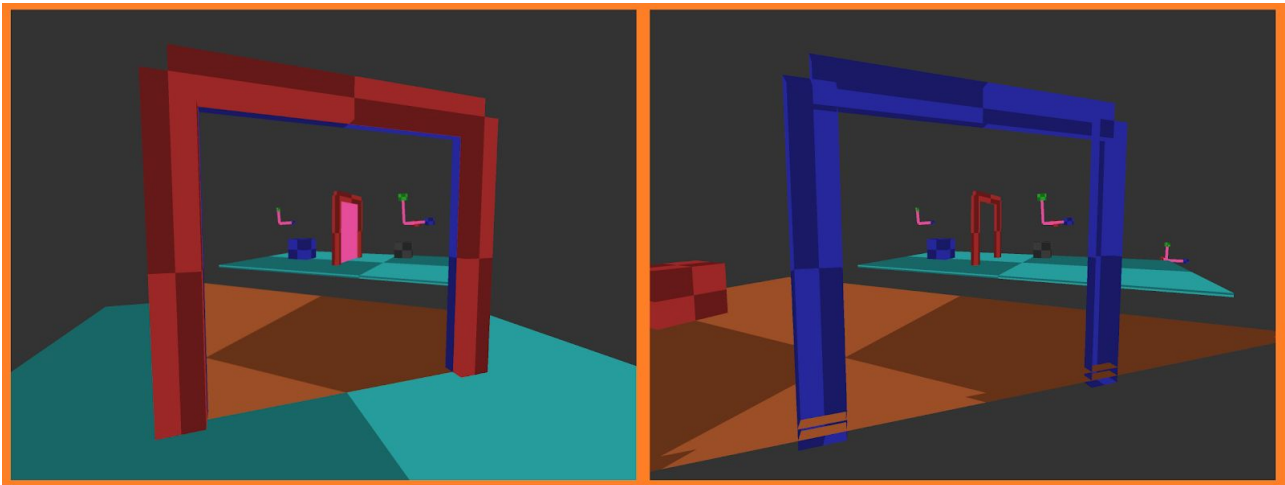
[Figura 6.4.1]: *Vista en primera persona (en perspectiva), en la que se muestra una recursión infinita de portales en línea recta. Simplemente se ha dispuesto un portal frente a otro, ambos orientados en la misma dirección y sentido. Cualquier ligera rotación tendría un gran impacto sobre el efecto final. En esta figura se observan simultáneamente las dos técnicas que se presentan en este apartado para producir recursión visual, combinadas en una sola para lograr un mejor resultado.*

La parte difícil de la recursión de portales es entender cómo generarla correctamente, su implementación como tal no es más complicada que otros conceptos anteriormente vistos. En vez de describirse directamente las formas más completas de generar esta recursión, es más didáctico avanzar poco a poco descartando alguna solución que parece buena a primera vista, pero que no funciona adecuadamente. Entender las razones por las que estas soluciones fallan o que les *falta*, probablemente lleve a deducir mejores métodos para generar esta recursión visual; incluso antes de ser presentados en esta memoria. Por ejemplo, en la figura [6.4.1] se combinan dos técnicas que por separado presentan algún error evidente.

La forma de recursión más simple, tiene lugar cuando desde el punto de vista de una cámara virtual se puede ver la superficie de su propio portal, pero como no existe una alineación suficiente se genera un único nivel de recursión. Cabe destacar que desde una cámara virtual, es imposible ver la superficie del portal contrario al suyo propio, esto se entiende mejor mediante un ejemplo: Desde la cámara virtual asignada al portal rojo, nunca se observa el portal azul, ya que dicha cámara se encuentra en el espacio local del portal azul y su función es ver la escena a través de *su superficie*. Luego en el resto de la escena, que se encuentra *tras su superficie*, es imposible que ésta aparezca de nuevo.

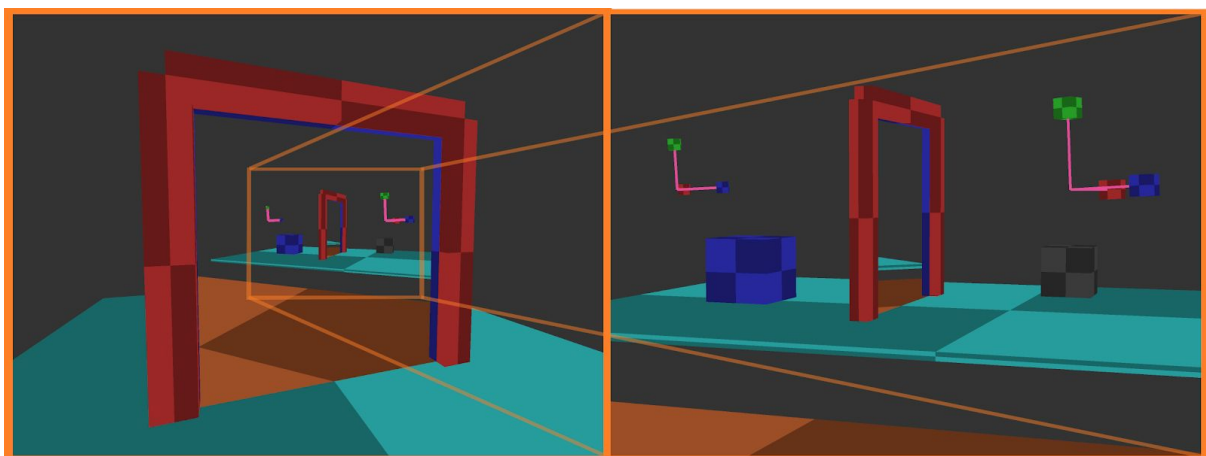


[Figura 6.4.2]: Esquema en vista cenital en el que se produce una recursión simple, de un único nivel de profundidad. Se han añadido representaciones (no a escala) de los espacios de vista de la cámara principal [derecha] y la cámara virtual del portal rojo [izquierda]. Debido a la posición y rotación de los portales, se produce una línea de visión a través de los mismos [flecha naranja] que permite ver el portal rojo sobre su propia superficie. Esto se debe a que el portal rojo es observado desde la vista de la cámara virtual, y esta vista posteriormente se plasma sobre la superficie del mismo portal.



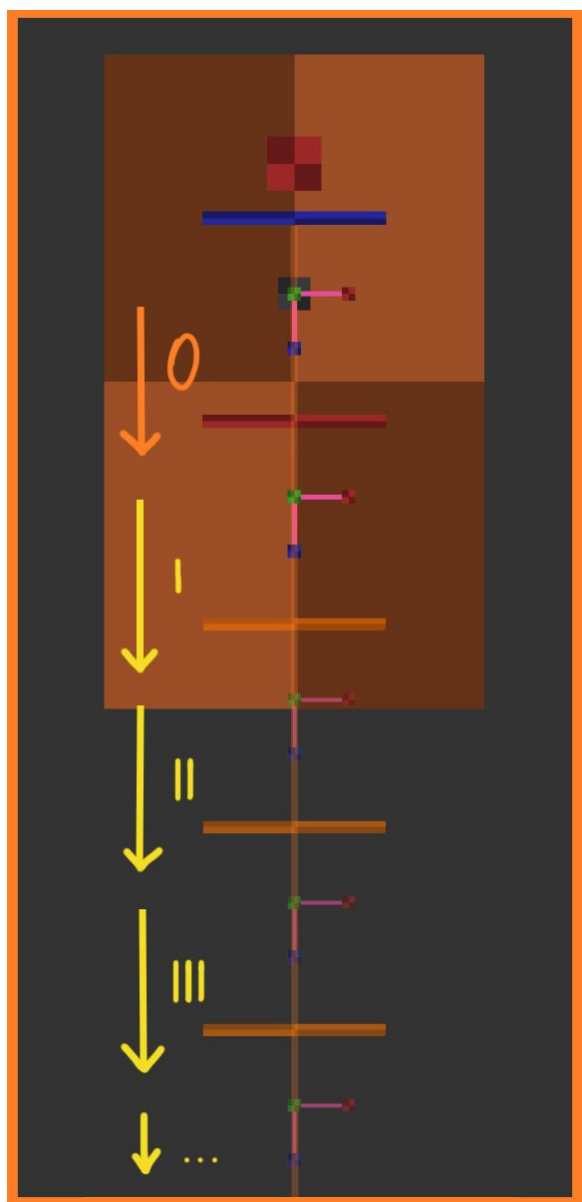
[Figura 6.4.3]: *Vistas en primera persona del esquema de la figura anterior [6.4.2]. En la imagen izquierda, se muestra la vista final de la cámara principal. En ella se puede observar a través de la superficie del portal rojo, el propio jugador y el portal rojo de nuevo; por ahora la superficie (y con ello la recursión) se encuentra desactivada (color rosa). En la imagen derecha, se muestra la vista de la cámara virtual. En esta ocasión se puede ver el plano de corte alineado con el portal en acción.*

Como se puede deducir, el efecto de la recursión siempre se encuentra totalmente contenido en la superficie del segundo portal, la cual es observada dentro del portal en primer plano (superficie de color rosa en la figura [6.4.3]). Luego lograr una recursión correcta se limita a sustituir dicha porción de la vista final por la imagen adecuada. Como ya fue mencionado, en caso de implementar los portales con texturas, esta imagen debe ser dibujada previamente para poder plasmarla en el resultado final. Concretamente, en toda recursión se debe comenzar a dibujar desde la imagen más profunda y retroceder hasta la vista principal. En cambio, en implementaciones basadas en *stencil-buffer*, las imágenes son las mismas pero la recursión se dibuja de menor a mayor profundidad, por ello es posible comenzar dibujando la vista principal y continuar si es necesario.



[Figura 6.4.4]: Vista en primera persona de la escena anterior, pero con la recursión activada. En la imagen derecha se muestra una ampliación.

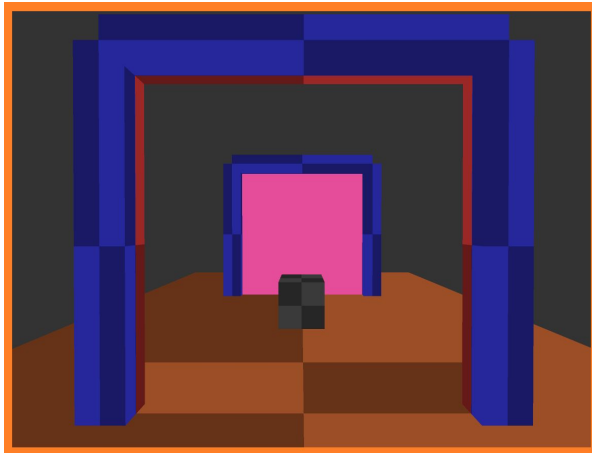
La imagen de cada paso recursivo se obtiene del renderizado de la escena desde una posición distinta. Pero al igual que con las cámaras virtuales, no basta con una simple posición sino que se necesita una transformación completa; cuyo cálculo además deriva del utilizado para dichas cámaras. En el fondo cuando se realiza recursión, se sustituye la transformación individual de una cámara virtual por una lista ordenada de ellas, calculadas o no previamente dependiendo de la implementación. Como toda operación recursiva, debe tener un límite definido, puede ser un número máximo de transformaciones o una condición más compleja como que el siguiente portal se encuentre ocluido.



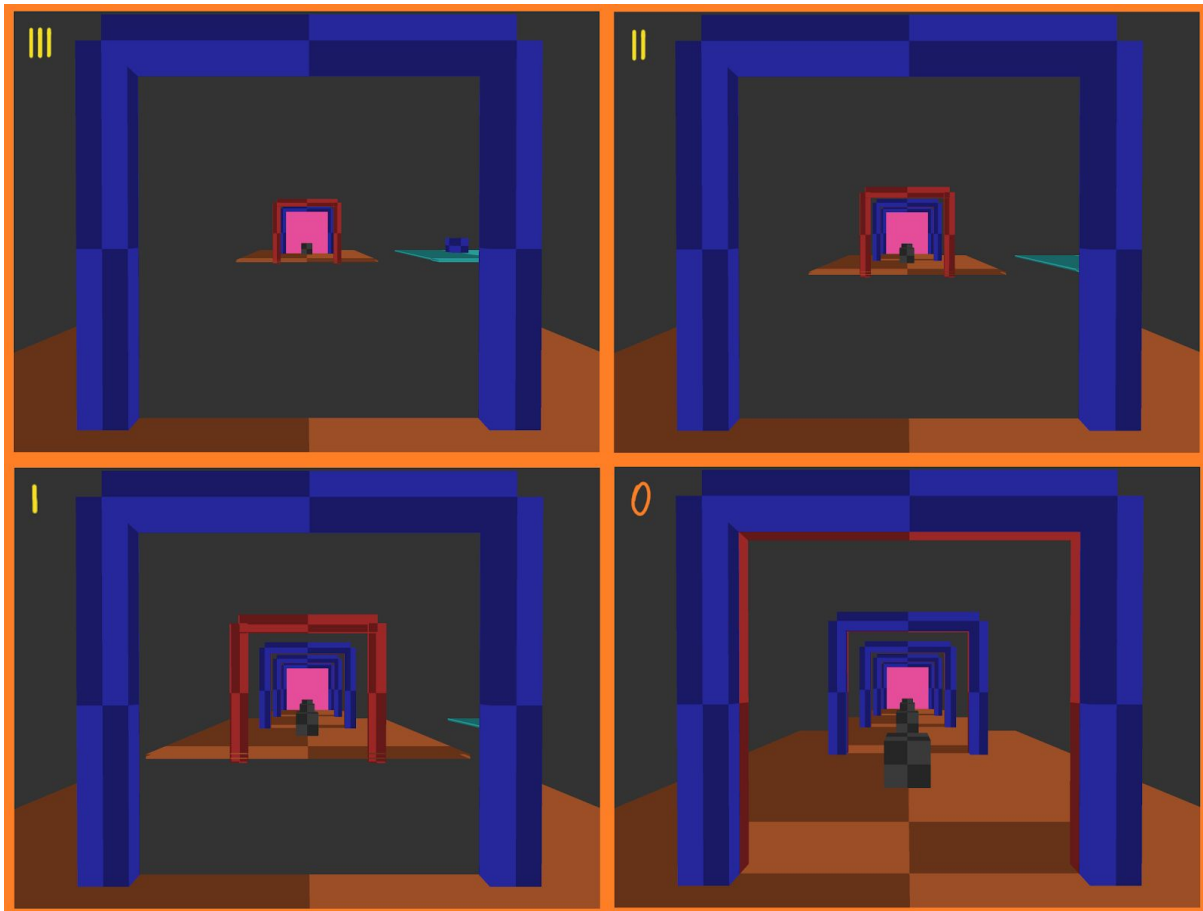
[Figura 6.4.5]: Vista cenital de una escena en la que se produce una recursión infinita (portales totalmente alineados). Se ha superpuesto un esquema con las posiciones recursivas de la cámara virtual azul. Todas estas transformaciones derivan de la original [flecha naranja 0], que fue explicada y utilizada previamente. Para obtener todas las posteriores se emplea el siguiente procedimiento [se referencian los colores mostrados en la figura y se supone que las cámaras virtuales son hijas del nodo del portal contrario (rojo)]:

1. Tomar la transformación del paso anterior, en espacio local del portal rojo.
2. Convertirla al espacio global de la escena.
3. Convertirla al espacio local del portal azul.
4. Guardarla como espacio local del portal rojo.

En esta escena la recursión sigue una línea recta, por ello las posiciones recursivas parecen sencillas, pero cualquier ligera rotación supondría un gran impacto.



[Figura 6.4.6]: Vista en primera persona de la escena mostrada en la figura anterior [6.4.5]. La recursión se encuentra desactivada, esta vista supone la transformación inicial [figura anterior, flecha naranja 0]. De nuevo, el efecto de toda recursión, por profunda que sea, se limita a afectar únicamente a la región coloreada de rosa.



[Figura 6.4.7]: Vista en primera persona que muestra la escena anterior [figura 6.4.5] y [6.4.6] con recursión activada. En esta implementación basada en texturas el orden de renderizado es inverso, por ello se comienza transformando la cámara virtual a la última y más alejada transformación calculada en la recursión, y luego se renderiza la escena completa (manteniendo planos de corte, etc) [III]. Después, en cada paso recursivo se transforma la cámara y se dibuja la escena de nuevo; en estos pasos se sobrescribe directamente toda la vista previa, menos la parte mostrada a través de la superficie del portal. En una implementación con stencil-buffer se comienza con la vista de la figura anterior [6.4.6], posteriormente en cada paso recursivo, se sustituye la superficie rosa por una vista de la escena que incluye el siguiente portal con nuevamente la superficie rosa.

```

// *** CALCULATE RECURSION TRANSFORMATIONS ***
// Recursion example for one portal
// + There must be a predefined recursion limit or conditions
glm::mat4 camMM = cam_->getModelMatrix(),
    ownPortalMM_inv = firstPortalData_->root_->getModelMatrix_Inversed(),
    linkedPortalMM_base = secondPortalData_->root_->getModelMatrix();

// Initial recursion is the basic one
// AGAIN: global player transformation to local space of its own portal
// + Stored as in local space of the linked portal (the camera is its child)
glm::mat4 combinedTrans = ownPortalMM_inv * camMM;

// IF the virtual camera is global, then add a final transformation to global space
//glm::mat4 combinedTrans = linkedPortalMM_base * ownPortalMM_inv * camMM;

// Texture based implementation: final transformations are applied in reverse order
// + Need for preallocated space for each transformation
// + Directly store them in reverse order for a later easier iteration
// + In this case, initial recursion is stored as the last one
recTrans_[recLimit_ - 1] = Transformation::getDecomposed(combinedTrans);
firstPortalData_->cam_->setLocalTrans(recTrans_[recLimit_ - 1]);

for (size_t i = 1; i < recLimit_; i++) {
    // Virtual camera transformation in local space of the linked portal
    // To global space, and then to local space of its own portal
    // Finally stored again as in local space of the linked portal (its child)
    combinedTrans = ownPortalMM_inv * linkedPortalMM_base * combinedTrans;

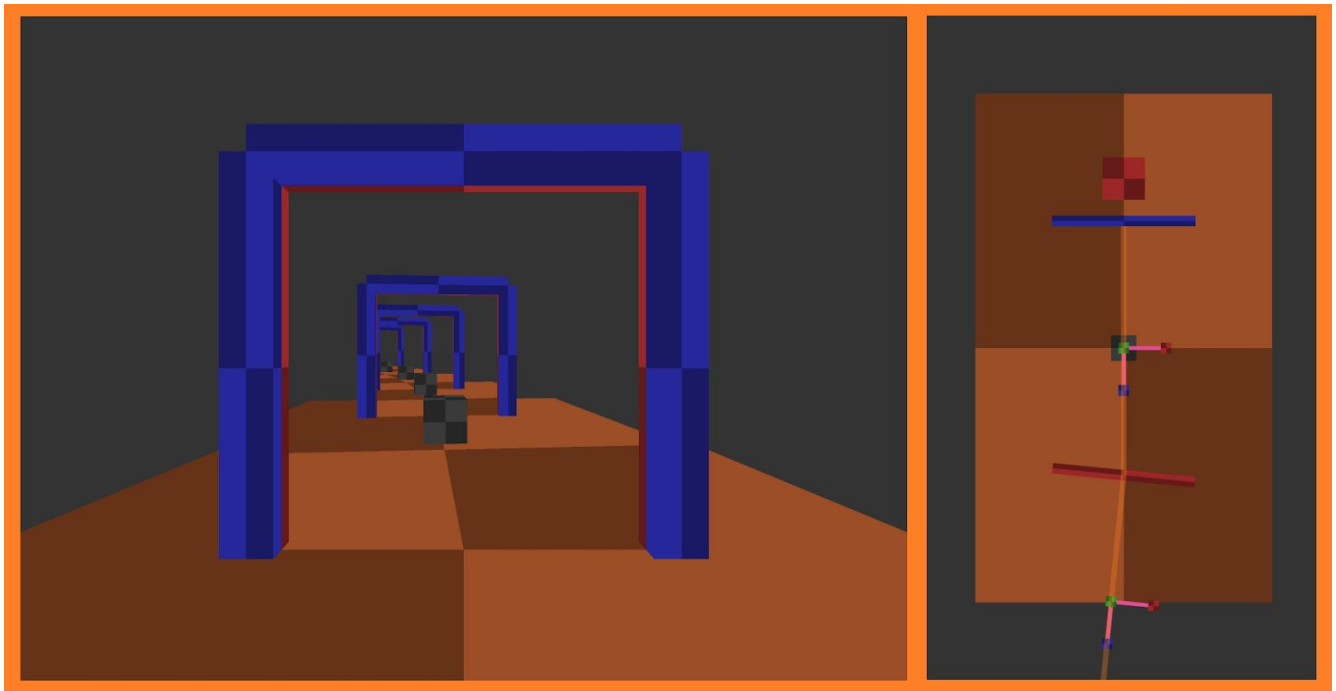
    //IF the virtual camera is global, the transformations order is switched
    //combinedTrans = linkedPortalMM_base * ownPortalMM_inv * combinedTrans;

    int renderOrderIndex = recLimit_ - i - 1; //to store in reverse order
    startIndex_ = renderOrderIndex; //in case of premature ending
    recTrans_[renderOrderIndex] = Transformation::getDecomposed(combinedTrans);
}

```

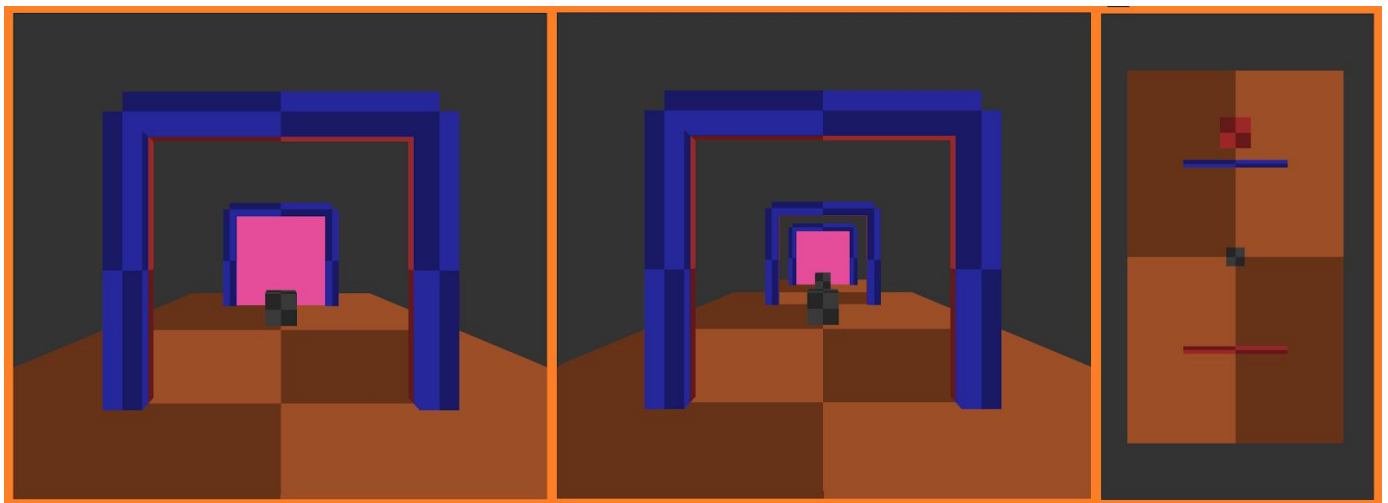
[Figura 6.4.8]: Muestra de la implementación particular de esta arquitectura del cálculo de las transformaciones recursivas de cada cámara virtual.

Cuando los portales se encuentran suficientemente alineados, por muy profunda que sea la recursión, la superficie *rosa* del último portal es visible. En algunas ocasiones, es posible dibujar tantas recursiones que el portal final queda fuera del plano lejano y consecuentemente no es observable. En caso de rotación, la superficie sale paso a paso del espacio de vista. El procedimiento actual para generar recursión es relativamente lento, computacionalmente hablando, e independientemente de la profundidad aplicada la superficie *rosa* final puede quedar a la vista. Es aceptable añadir algún tipo de efecto para disimular esta superficie [además de no colorearla llamativamente para destacar, cómo en estas figuras], por ejemplo uso de niebla en la distancia para que no desaparezca de golpe. Pero una mejor solución, es tratar de eliminar esta superficie y simplificar la recursión en el proceso.

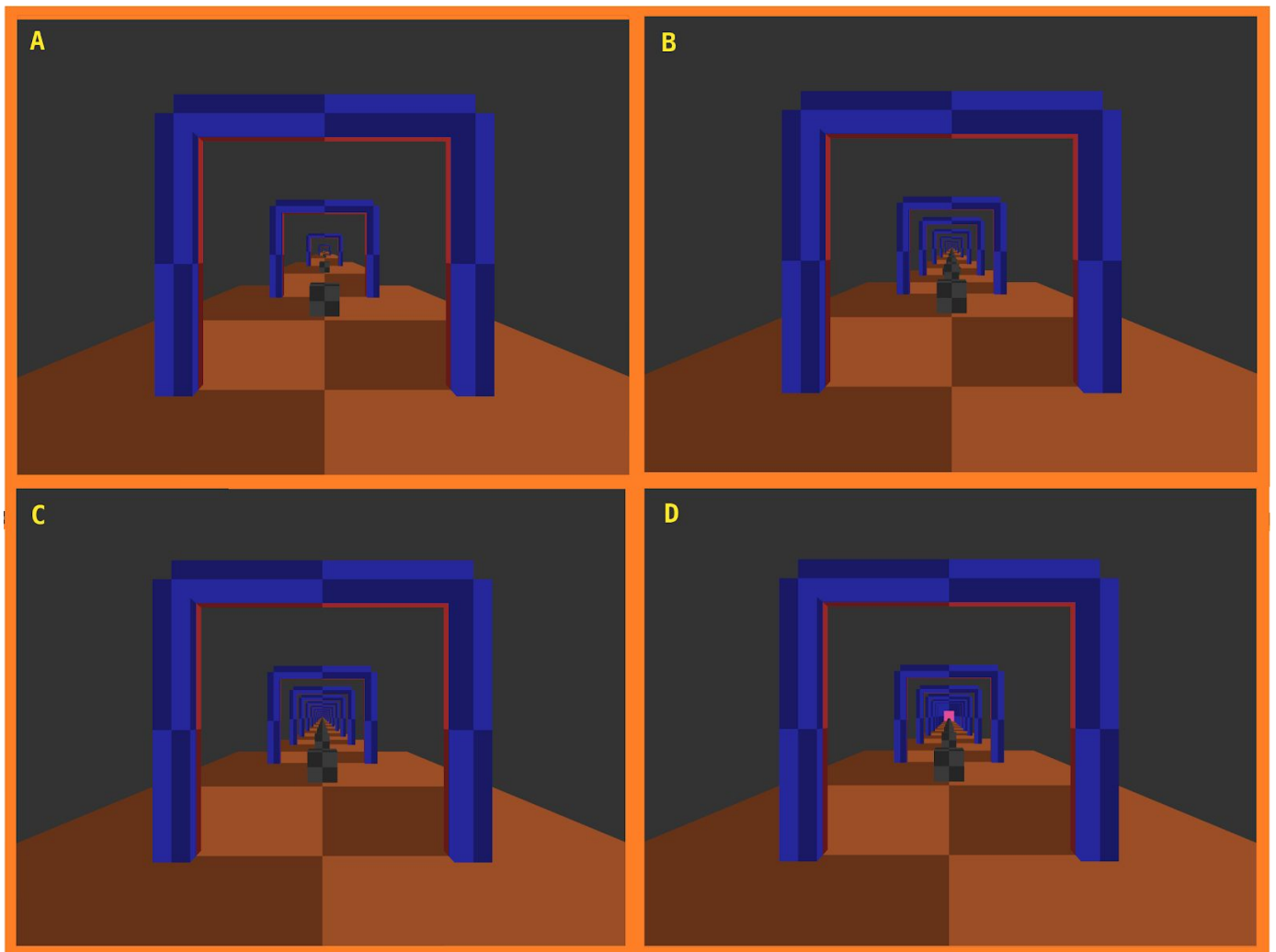


[Figura 6.4.5]: *En la imagen izquierda, se muestra una vista en primera persona de una escena con una recursión ligeramente rotada. En la imagen derecha, se puede observar un esquema cenital en el que se aprecia la cantidad de rotación del portal. En cada paso recursivo se aplica la misma cantidad de rotación, respecto a la transformación anterior. Aunque la rotación sea mínima, con 5 recursos basta para ocultar la superficie rosa final.*

Luego los casos de interés son las recursiones suficientemente rectas (prácticamente infinitas), y el objetivo es eliminar el plano rosa y limitar la profundidad de la recursión. En estos casos, la recursión tiene una propiedad peculiar: la imagen que se añade en cada paso recursivo es muy parecida a la inicial, pero reducida en escala. Se puede aprovechar esta propiedad para renderizar la escena recursiva un número reducido de veces y después sustituir el plano rosa final por la imagen producida reescalada. Por ejemplo, se realiza una recursión de tres niveles de profundidad y dicho plano rosa se texturiza con una versión modificada de la imagen observada en el portal original (que contiene la superficie rosa): recortada, reescalada y copiada múltiples veces hasta que esta parte rosa es inapreciable; si sólo se copia una vez, es posible que la superficie problemática aún siga presente en la imagen, aunque más pequeña. A mayor número de escenas recursivas reales, mayor fidelidad y mejor resultado visual, pero también superior coste de cómputo. Entonces, la clave se encuentra en balancear la combinación de estas dos técnicas. A esto hay que añadir, que en la segunda técnica, tanto la implementación como la propia modificación de imágenes, son bastante libres y flexibles.

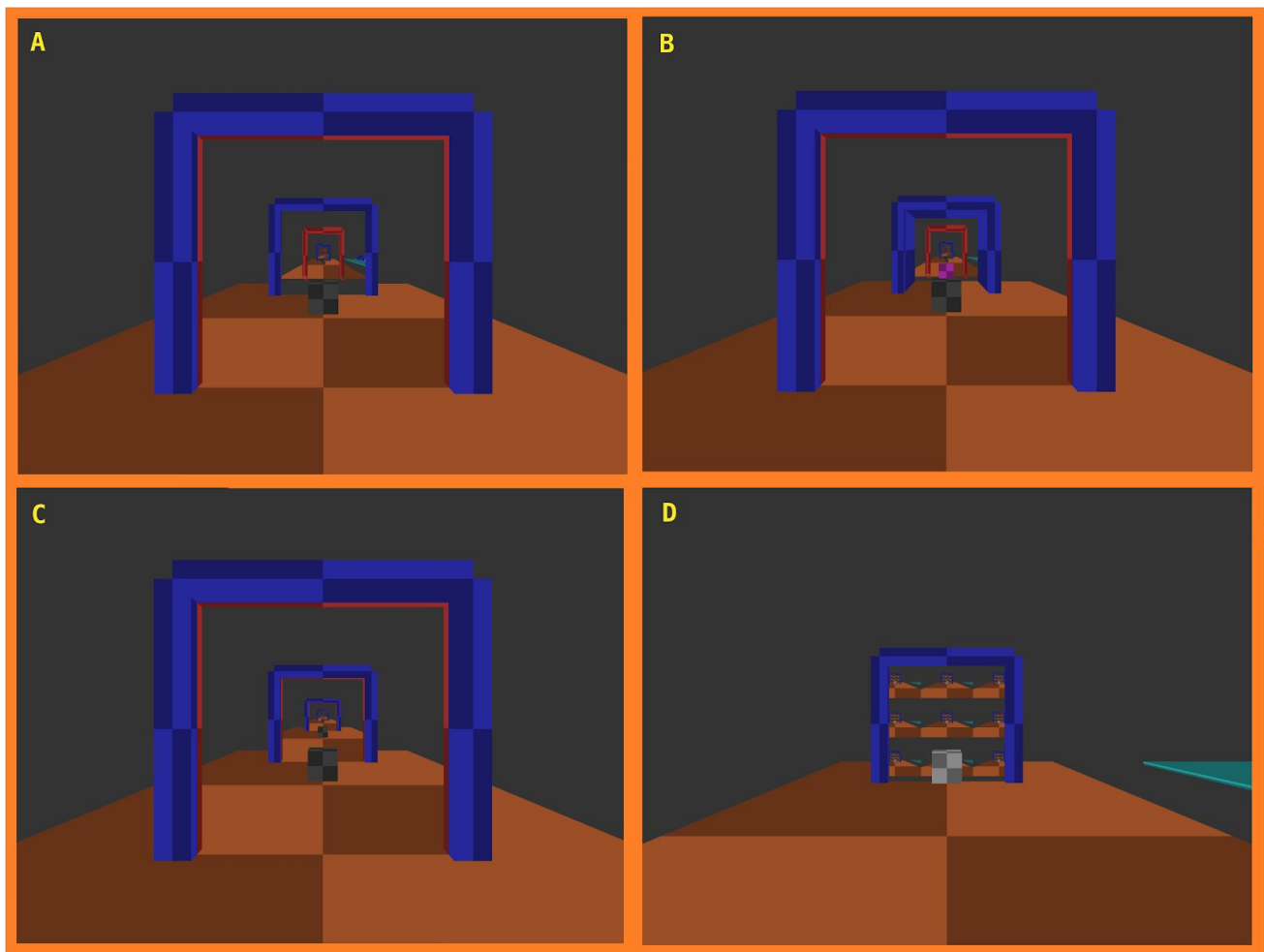


[Figura 6.4.10]: *En la vista izquierda, se muestra en primera persona una escena sin recursión. En la vista derecha, se muestra la misma escena, pero con una recursión de un nivel de profundidad. Se puede apreciar como la imagen que ha sustituido la superficie rosa, es prácticamente la misma que la textura del portal original reescalada. Se ha añadido una vista cenital de la escena como referencia.*



[Figura 6.4.11]: *Vistas en primera persona de la escena de la figura anterior [6.4.10]. Se ofrece una comparación entre diferentes resultados obtenidos de la combinación de ambas técnicas previamente expuestas. En la vista [D] no se emplea escalado, sino que se realiza una recursión real de 12 niveles de profundidad, esta imagen es la más fiel a la realidad. En la vista [A] en cambio, no se emplea ningún nivel de recursión y directamente se aplica escalado; en este caso el resultado no es aceptable ya que dista demasiado de [D]. En la vista [B] se emplea un único nivel de recursión, el resultado es satisfactorio. En la vista [C] se emplean dos niveles de recursión y el resultado es muy parecido a [D]. La vista [C] es un sustituto adecuado en el que se emplea tan sólo una pequeña porción de las recursiones, y además concluye la recursión en el infinito (no se muestra la superficie rosa). Dependiendo del nivel de detalle deseado, la vista [B] también es suficiente.*

En la implementación de esta arquitectura se aprovecha el uso de texturas para la técnica de reescalado. Concretamente, se utiliza la imagen del portal renderizada en el ciclo anterior; en vez de borrar la textura del *framebuffer*, se reutiliza sin necesidad de volver a ser dibujada. Debido al reescalado, la diferencia en la imagen de un ciclo a otro es tan mínima que no es apreciable. Esto incluso permite dibujar cada paso recursivo en el orden inverso esperado; en vez de modificar la imagen posteriormente para eliminar la superficie rosa, se sustituye directamente aplicando dicha textura al portal de la última recursión. En esta implementación, la textura no se escala *realmente*, sino que se asigna a su portal final de una forma especial. Tampoco se copia la superficie rosa hasta desaparecer, en cambio, se permite que simplemente la sucesión de la textura ciclo a ciclo (reutilizada del ciclo anterior), reduzca su escala a la mitad cada vez hasta desaparecer. Por ejemplo, en la vista [B] de la figura [6.4.11] se muestra el resultado tras unas pocas vueltas de bucle, incluso partiendo de un único nivel de recursión. Al inicio de la ejecución el plano rosa es visible pero a cada ciclo se dobla su profundidad hasta desaparecer, y así se mantiene hasta que se desactiva y reactiva la propia recursión. En la aplicación, todas estas optimizaciones se pueden ajustar, en este caso por una insignificante pérdida de fidelidad final se reduce considerablemente el coste computacional del proceso recursivo. Esto es común en videojuegos, donde en muchas ocasiones basta con que un efecto sea “*suficientemente bueno*”, y es mucho más importante que no sea costoso.



[Figura 6.4.12]: *Vistas en primera persona de la escena de las figuras anteriores [6.4.10] y [6.4.11]. Esta figura muestra brevemente cómo se asigna la textura al último portal de la recursión. No se utiliza ningún nivel de recursión para que el efecto sea más evidente. En la vista [A], se asigna la textura incorrectamente completa, al igual que en el final del apartado [6.1]. En la vista [B] la textura se asigna como en el resto de portales, esto también produce un resultado inválido, ya que en este caso la imagen depende del ciclo anterior. Es inútil asignar exactamente la misma porción de pantalla del ciclo anterior al actual porque es un proceso destructivo, se sobrescribe la información al dibujar en la misma posición. En la vista [C] se muestra como asignar correctamente la textura. Este éxito se consigue asignando como coordenadas de textura, las coordenadas en pantalla de los vértices del portal conectado (en esta escena el rojo). En la vista [D], se muestra como curiosidad que esta asignación de textura necesita como mínimo un nivel de recursión; cosa que esta vista no tiene, y por ello al aproximarse a la superficie, las coordenadas de textura del portal rojo se salen de pantalla y se asignan indefinidamente.*

6.5. Conclusión

Se ha terminado esta explicación sobre el funcionamiento y la implementación de los portales. Se ha tratado de ser suficientemente abstracto para que sean reproducibles con facilidad sobre la arquitectura de cualquier otro motor. Concretamente, la memoria se ha centrado en el apartado gráfico, para el que se han visto varios métodos de implementación; incluso dentro del seleccionado, existen múltiples opciones de configuración, como por ejemplo las diferentes técnicas recursión.

La aplicación de este proyecto se ha centrado en el desarrollo de una escena totalmente interactiva en la que aplicar las técnicas de implementación de portales descritas, y que además se ha utilizado para generar con facilidad las diversas imágenes mostradas como ejemplos en las figuras de la memoria (sin tocar el código fuente). Esta implementación ha resultado esencial como apoyo experimental durante la realización del proyecto, en su repositorio [1] se encuentra el código que completa la información de esta memoria. Entre sus funcionalidades en tiempo real se incluyen: movimiento en primera persona por la escena, transformación de todos los portales, intercambio de cámara principal, vista cenital, zoom, bloqueo de vista, efectos de post-procesado, activación y desactivación de distintos objetos, ajustes de modo de asignación de coordenadas de texturas del portal, ajustes del comportamiento del prisma del portal, ajustes de copia del cuerpo del jugador, ajuste de plano de proyección oblicua, ajuste de cantidad y modo de recursión, etc. La información relacionada con el control de estas funcionalidades se encuentra en el [Apéndice C].

Respecto al diseño de los procedimientos explicados, la mayoría proviene de la mencionada saga de *Portal* (el caso de estudio), aunque ninguna línea de código es accesible públicamente. La solución que se plantea en el apartado [6.2] referente al uso de portales con un determinado grosor, proviene de una implementación sencilla de portales en *Unity* [68], encontrada durante la investigación. El uso de proyecciones con planos de corte oblicuos en el apartado [6.3], no es necesario y no se usan en *Portal*, pero como ya fue comentado resulta didáctico. La combinación final de técnicas recursivas del apartado [6.4] es muy flexible y configurable, y en este caso es un diseño propio.

7. Conclusiones y trabajo futuro

En este conciso capítulo final se presentan las conclusiones del proyecto respecto a sus objetivos y plan de trabajo. También se añaden posibles ideas de trabajo futuro.

7.1. Conclusiones

Una vez finalizado el proyecto es un buen momento para reflexionar y realizar una valoración sobre el trabajo completado y lo aprendido durante el mismo. Desde el punto de vista de los objetivos listados en el apartado [1.2], **menos los dos últimos**, todos se han cumplido satisfactoriamente a tiempo. Este resultado se debe a que desde el principio, en el plan de trabajo, se ha tenido en cuenta la posibilidad de no poder lograr todos los objetivos, y por ello se han perseguido por orden de prioridad.

Respecto a los **objetivos completados**, alguno ha resultado más complicado de lo anticipado. El trabajo relacionado con el estudio de *OpenGL* [2] y la implementación del motor [3] ha costado bastante más de lo esperado, pero igualmente estoy satisfecho con el aprendizaje. La cantidad de tiempo invertida en la investigación [1] y el desarrollo [5] de los portales ya había sido prevista, aunque la implementación final de la recursión ha llevado más tiempo ya que ha ido mejorando progresivamente. Creo que el nivel de detalle en la redacción de la memoria [6, 7, 8] ha sido más elevado del intencionado inicialmente, y con ello el volumen de páginas y trabajo, pero estoy muy contento con el resultado. La interactividad de la escena implementada [4] ha crecido continuamente junto con la redacción de contenidos, y ha llegado a un nivel de interacción y configuración más que destacable. La recursión de los portales ha sido especialmente difícil de explicar [7] y ha requerido una gran cantidad de figuras.

No se han logrado los objetivos relacionados con la implementación de mejoras en el motor gráfico [9] y el desarrollo de escenas más interesantes para los portales [10]. Estos objetivos de menor prioridad, han resultado descartados por falta de tiempo y *necesidad*. El motor gráfico actual ha permitido generar todas las imágenes para la memoria, y las propias texturas

sencillas sin iluminación aportan claridad y simpleza al resultado. Asimismo, la importación de modelos no ha sido estimada y se han utilizado objetos de geometría sencilla. La escena final que se centra en la interactividad y utilidad, permite configurar y mostrar cada una las características de los portales en tiempo real. No se usan los portales de formas *sorprendentes*, pero las propias opciones de configuración son bastante más didácticas e *interesantes* de lo planeado inicialmente.

7.2. Trabajo futuro

En este apartado se describen posibles ideas que añaden más valor al proyecto. Además de los dos objetivos no cumplidos, a lo largo de la realización de este proyecto han surgido otras ideas interesantes. A continuación se listan sin un orden particular:

- *Objetivo previo:* Implementar mejoras sobre el motor gráfico, principalmente soporte para iluminación y para importación de modelos o escenas completas.
- *Objetivo previo:* Desarrollar escenas más sorprendentes, que empleen los portales de formas más interesantes. Por ejemplo, con más de un par de portales.
- Desarrollo de una interfaz gráfica que facilite la interacción con la escena. Esto permitiría al usuario tener una experiencia mejor, sin necesidad de conocer los controles.
- Desarrollo de una implementación alternativa basada en *stencil-buffer*, con la capacidad de poder activarla en tiempo real. Esto permitiría poder realizar una comparación más directa entre ambos métodos de implementación.
- Desarrollo de diversas optimizaciones, éstas podrían ser interesantes para casos en los que se simulen escenas relativamente complejas. Por ejemplo: detección de portales fuera del espacio de pantalla para agilizar el proceso de renderizado o detección de ausencia de superposición entre portales en espacio de pantalla para agilizar el proceso de recursión.
- Creación de un *Shader* que optimice la gestión de las texturas de los portales. En vez de guardar una textura con la escena completa y posteriormente asignarla al portal correctamente, quizás sea posible guardarla previamente de forma *optimizada* para que la asignación sea directa (que no requiera un proceso especial).

Bibliografía

- [1] Diego Mateos Arlanzón. TFG - MOTOR GRÁFICO CON PORTALES PARA SIMULAR ESCENAS 3D NO EUCLIDIANAS. Repositorio del proyecto. https://github.com/dimateos/TFG_Portals, 2020
- [2] Portal: No Escape. Live Action Short Film by Dan Trachtenberg. <https://www.youtube.com/watch?v=4drucglA6Xk>, 2011
- [3] Portal. Selling point reference. <https://store.steampowered.com/app/400/Portal>, 2007
- [4] Computer History Museum (Mountain View, CA). The Utah Teapot. <https://www.computerhistory.org/revolution/computer-graphics-music-and-art/15/206>, 2020
- [5] Unity Technologies. Unity official webpage. <https://unity.com/es>, 2020
- [6] Epic Games, Inc. Unreal Engine official webpage. <https://www.unrealengine.com/en-US/>, 2020
- [7] GIGABYTE productos. Tarjeta gráfica GeForce RTX 2060. <https://www.gigabyte.com/es/Graphics-Card/GV-N2060GAMING-OC-6GD#kf>, 2018
- [8] Nvidia Corporation. Nvidia official webpage. <https://www.nvidia.com/es-es/about-nvidia/>, 2020
- [9] GIGABYTE. Gigabyte official webpage. <https://www.gigabyte.com/About>, 2020
- [10] Khronos Group Inc. OpenGL - The Industry's Foundation for High Performance Graphics. <https://www.opengl.org/>, 2020
- [11] Khronos Group Inc. Khronos official webpage. <https://www.khronos.org/>, 2020
- [12] OpenGL Wiki contributors. Language bindings. https://www.khronos.org/opengl/wiki/Language_bindings, 2020
- [13] Standard C++ Foundation. Standard C++. <https://isocpp.org/>, 2020
- [14] Khronos Group Inc. Vulkan Overview. <https://www.khronos.org/vulkan/>, 2020
- [15] Khronos Group Inc. Vulkan Overview slides 2016. <https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf>, 2016
- [16] Microsoft. Getting started with Direct3D. <https://docs.microsoft.com/en-us/windows/win32/getting-started-with-direct3d>, 2018

- [17] Nvidia Gameworks. Transitioning from OpenGL to Vulkan. <https://developer.nvidia.com/transitioning-opengl-vulkan>, 2016
- [18] UNIVERSIDAD COMPLUTENSE DE MADRID (UCM). FACULTAD DE INFORMÁTICA (FDI). Ficha docente de la asignatura IG I. http://web.fdi.ucm.es/UCMFiles/pdf/FICHAS_DOCENTES/2017/1362.pdf, 2017-2018
- [19] OpenGL Wiki contributors. History of OpenGL. https://www.khronos.org/opengl/wiki/History_of_OpenGL, 2020
- [20] UNIVERSIDAD COMPLUTENSE DE MADRID (UCM). FACULTAD DE INFORMÁTICA (FDI). Ficha docente de la asignatura IG II. http://web.fdi.ucm.es/UCMFiles/pdf/FICHAS_DOCENTES/2017/1372.pdf, 2017-2018
- [21] OGRE. OGRE - Open Source 3D Graphics Engine. <https://www.ogre3d.org/>, 2020
- [22] OpenGL Wiki contributors. Shader. <https://www.khronos.org/opengl/wiki/Shader>, 2019
- [23] OpenGL Wiki contributors. Core Language (GLSL). [https://www.khronos.org/opengl/wiki/Core_Language_\(GLSL\)](https://www.khronos.org/opengl/wiki/Core_Language_(GLSL)), 2019
- [24] Joey de Vries (<https://twitter.com/JoeyDeVries>). Learn OpenGL. <https://learnopengl.com/>, 2020
- [25] Joey de Vries. Kendall & Welling publishing. Learn OpenGL - Graphics Programming. https://learnopengl.com/book/book_pdf.pdf, 2020
- [26] Joey de Vries. learnopengl.com code repository. <https://github.com/JoeyDeVries/LearnOpenGL>, 2020
- [27] Creative Commons. Attribution-NonCommercial 4.0 International. <https://creativecommons.org/licenses/by-nc/4.0>, 2016
- [28] Creative Commons. Attribution 4.0 International. <https://creativecommons.org/licenses/by/4.0>, 2016
- [29] Alexander Overvoorde. Modern OpenGL Guide. <https://open.gl/>, 2019
- [30] Alexander Overvoorde. open.gl content repository. <https://github.com/Overv/Open.GL>, 2019
- [31] Creative Commons. Attribution-ShareAlike 4.0 International. <https://creativecommons.org/licenses/by-sa/4.0/>, 2016

- [32] OpenGL Wiki contributors. OpenGL Wiki. <https://www.khronos.org/opengl/wiki/>, 2018
- [33] OpenGL Mathematics. GL 0.9.9 API documentation, <https://glm.g-truc.net/0.9.9/api/modules.html>, 2020
- [34] Simple DirectMedia Layer. SDL 2.0 wiki frontpage. <https://wiki.libsdl.org/FrontPage>, 2018
- [35] GLFW. GLFW - An OpenGL library. <https://www.glfw.org/>, 2020
- [36] glad. GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs. <https://github.com/Dav1dde/glad>, 2020
- [37] OpenGL Wiki contributors. OpenGL Loading Library. https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library, 2019
- [38] OpenGL Mathematics (GLM). A C++ mathematics library for graphics programming. <https://glm.g-truc.net/0.9.9/index.html>, 2020
- [39] stb_image. Image loading/decoding from file/memory: JPG, PNG, TGA, BMP, PSD, GIF, HDR, PIC. <https://github.com/nothings/stb>, 2020
- [40] Microsoft. IDE de Visual Studio. <https://visualstudio.microsoft.com/es/>, 2020
- [41] Linus Torvalds. Git, free and open source. <https://git-scm.com/>, 2020
- [42] GitHub, Inc. About. <https://github.com/about>, 2020
- [43] Update method. Robert Nystrom, Game Programming Patterns. <https://gameprogrammingpatterns.com/update-method.html>, 2014
- [44] Game loop. Robert Nystrom, Game Programming Patterns. <https://gameprogrammingpatterns.com/game-loop.html>, 2014
- [45] Delta time. Cristian Barrio (BYC), Parallelcube. <https://www.parallelcube.com/es/2017/10/25/por-que-necesitamos-utilizar-delta-time/>, 2017
- [46] Event-Subscriber pattern. Alexander Shvets (refactoring.guru), Dive Into Design Patterns. <https://refactoring.guru/design-patterns/observer>, 2020
- [47] Paul E. Black and Algorithms and Theory of Computation Handbook, CRC Press LLC, 1999. Tree. Dictionary of Algorithms and Data Structures. <https://xlinux.nist.gov/dads/HTML/tree.html>, 2017

- [48] Wikipedia contributors. Tree traversal. In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Tree_traversal&oldid=968976983, 2020
- [49] Sumanta Guha; Computer Graphics through OpenGL; Segunda edición, CRC Press, 2015
- [50] Peter Shirley, Steve Marschner; Fundamentals of Computer Graphics; Third Edition, CRC Press, 2009
- [51] OpenGL Wiki contributors. Tessellation. <https://www.khronos.org/opengl/wiki/Tessellation>, 2019
- [52] Department of Mathematics. University of Maryland. Gimbal Lock. http://www.math.umd.edu/~immortal/MATH431/lecturenotes/ch_gimballock.pdf, 2018
- [53] Wolfram mathworld. Quaternion. <https://mathworld.wolfram.com/Quaternion.html>, 2020
- [54] Blender Documentation Team. The Blender 2.90 Manual. Licensed under CC-BY-SA v4.0. [https://docs.blender.org/manual/en/latest/editors/3dview/navigate/ views.html](https://docs.blender.org/manual/en/latest/editors/3dview/navigate/views.html), 2020
- [55] Techopedia. First Person Shooter (FPS). <https://www.techopedia.com/definition/241/first-person-shooter-fps>, 2011
- [56] Valve Corporation. Official webpage. <https://www.valvesoftware.com/es/>, 1996-2020
- [57] Portal 2. Official webpage. <https://www.thinkwithportals.com/>, 2011
- [58] Narbacular Drop. Overview from official webpage. <http://www.nuclearmonkeysoftware.com/narbaculardrop.html?overview.html>, 2004-2008
- [59] DigiPen Institute of Technology. Official webpage. <https://www.digipen.edu/>, 2020
- [60] Antichamber. Official webpage. <http://www.antichamber-game.com/>, 2013
- [61] Narbacular Drop available documents. Including Technical Design, Game Design and Postmortem. <http://www.nuclearmonkeysoftware.com/narbaculardrop.html>, 2005
- [62] DigiPen Institute of Technology. Narbacular Drop Technical Design Document. http://www.nuclearmonkeysoftware.com/documents/narbacular_drop_technical_design_document.pdf, 2004

- [63] Portal Problems - Lecture 11 - CS50's Introduction to Game Development 2018. CS50, Harvard University. <https://www.youtube.com/watch?v=ivyseNMVt-4>, 2018
- [64] Creative Commons. Attribution-NonCommercial-ShareAlike 4.0 International Public License. <https://creativecommons.org/licenses/by-nc-sa/4.0/>, 2016
- [65] Game Developers Conference (GDC). Portal 2: Creating a Sequel to a Game That Doesn't Need One. <https://www.youtube.com/watch?v=BYFvwbbY2YM>, 2016
- [66] The Portal Wiki. Unofficial wiki. https://theportalwiki.com/wiki/Main_Page, 2020
- [67] Geoff Keighley. Documentary: The Final Hours of Portal 2. <http://www.thefinalhoursofportal2.com/>, 2011
- [68] Sebastian Lague. Coding Adventure: Portals in Unity. Licensed under MIT License. <https://github.com/SebLague/Portals>, 2020
- [69] Wolfram mathworld. Plane. <https://mathworld.wolfram.com/Plane.html>, 2020
- [70] Eric Lengyel. Terathon Software. Oblique View Frustum Depth Projection and Clipping. Published in Journal of Game Development, Vol. 1, No. 2. <http://www.terathon.com/lengyel/Lengyel-Oblique.pdf>, 2005

Apéndice A. Introduction [EN]

In this brief chapter, the motivations behind the project, its specific objectives and the followed work plan are all presented. The complete memory structure is also schematically explained.

The associated source code can be found at: https://github.com/dimateos/TFG_Portals

A.1. Motivation

Portals are an interesting phenomenon for which examples can be found in various productions from all kinds of industries: animation, film, video games, etc. Among these, the portals used in videogames stand out, since their properties are normally rigorously defined and simulated in real time. But unfortunately, its technical implementation is complex and the information available about it is limited and scattered (and totally non-existent in Spanish). From this fact arises the desire to carry out extensive research in order to be able to build my own implementation. Furthermore, the intention in this project is to start from scratch and create a simple graphics engine; this allows me to expand my knowledge of computer graphics and engines in general.

The final purpose of the project is to solve the lack of concrete information initially encountered. This is intended to be achieved by writing a document that explains the inner workings and the implementation of the portals in a simple graphics engine. So this dissertation itself, together with the associated source code of my implementation, would form a public resource that stands out for being complete, rigorous and self-contained. To make the explanation as accessible and didactic as possible, even the fundamentals of computer graphics and the technologies used are introduced so that a person outside the subject, if they wish, can also follow the technical part of the exposition. Regarding my own implementation, it is detailed in full, so that it may be consulted in a useful way. And finally, the concrete explanation regarding the portals remains abstract enough to be easily reproducible in the architecture of any other engine.

A.2. Goals and work plan

The objectives of this project can be listed in order of priority. The probability of being unable to complete all of them due to a higher than anticipated workload is taken into account. This same order has been followed as a work plan:

- Study how portals and existing implementation methods work.
- Study the fundamentals of computer graphics required to implement them.
- Develop a simple reusable graphics engine with sufficient logic capabilities.
- Implement an interactive 3D scene as a base for the development.
- Implement the portals progressively, starting with its most basic properties, to the most complex ones.
- Write the part of the dissertation referring to the fundamentals of computer graphics, to make it more accessible, and to ease the later explanation.
- Write the part of the dissertation referring to the complete inner workings and implementation of the portals, as didactic and useful as possible.
- Write the part of the dissertation referring to the architecture of the engine itself, so that it is possible to consult the source code in a productive way.
- Implement improvements to the graphics engine, mainly support for lighting and for importing models or complete scenes.
- Develop more impressive scenes, using the portals in surprising and interesting ways.

A.3. Structure of the memory

This document tries to be very accessible and therefore part of it may be redundant information for a person who already knows the subject. Specifically, chapters [2] and [4] referring to computer graphics, can be read over if an explanation of their state of the art and fundamentals is not required. Chapter [3] briefly details the architecture of the engine itself, if this is not of interest, it may be skipped and consulted in case of doubt. Finally, the extensive chapters [5] and [6] explain the state of the art of the portals and their inner workings and implementation completely.

Apéndice B. Conclusions and future work [EN]

This concise final chapter presents the conclusions of the project regarding its objectives and work plan. Possible ideas for future work are also added.

B.1. Conclusions

Once the project is finished, it is a good time to reflect and make an assessment of the work completed and what has been learned during itself. From the point of view of the objectives listed in section [1.2], **except the last two**, all have been fulfilled satisfactorily on time. This result is due to the fact that from the beginning, in the work plan, the possibility of not being able to achieve all the objectives was already taken into account, and therefore they have been pursued in order of priority.

Regarding the **completed objectives**, some have been more complicated than anticipated. The work related to the study of *OpenGL* [2] and the implementation of the engine [3] has taken a lot longer than expected, but I am still satisfied with the learning. The amount of time invested in research [1] and development [5] of the portals had already been anticipated, although the final implementation of the recursion has taken longer as it has progressively improved. I think that the level of detail in the writing of the dissertation [6, 7, 8] has been higher than initially intended, and with it the volume of pages and work, but I am very happy with the result. The interactivity of the implemented scene [4] has grown continuously along with the writing of content, and has reached a level of interaction and configuration more than remarkable. Portal recursion has been especially difficult to explain [7] and has required a large number of figures.

The objectives related to the implementation of improvements in the graphics engine [9] and the development of more interesting scenes for the portals [10] have not been achieved. These lower priority objectives have been discarded due to lack of time and *necessity*. The current graphics engine has managed to generate all the images for the dissertation, and the simple textures without lighting themselves add clarity and simplicity to the result. Likewise, the importing of models has not been estimated and simple geometry objects have been used.

The final scene that focuses on interactivity and utility, allows you to configure and display each of the characteristics of the portals in real time. The portals are not used in *surprising* ways, but the configuration options themselves are much more didactic and *interesting* than initially planned.

B.2. Future work

This section describes possible ideas that add more value to the project. In addition to the two unfulfilled objectives, other interesting ideas have emerged throughout the execution of this project. Here they are listed below in no particular order:

- *Previous objective:* Implement improvements to the graphics engine, mainly support for lighting and for importing models or complete scenes.
- *Previous objective:* Develop more impressive scenes, using the portals in surprising and interesting ways. For example, with more than a couple of portals.
- Development of a graphical interface that facilitates interaction with the scene. This would allow the user to have a better experience, without needing to know the controls.
- Development of an alternative implementation based on stencil-buffer, with the ability to activate it in real time. This would allow a more direct comparison between both implementation methods.
- Development of various optimizations, these could be interesting for cases in which relatively complex scenes are simulated. For example: detection of portals outside the screen space to speed up the rendering process or detection of the absence of overlapping between portals in screen space to speed up the recursion process.
- Creation of a Shader that optimizes the management of portals textures. Instead of saving a texture with the entire scene and then assigning it to the portal correctly, it may be possible to pre-save it in an *optimized* way so that the assignment is straightforward (requiring no special processing).

Apéndice C. Controles de la aplicación [ES]

A continuación se muestra un esquema con los controles disponibles para interactuar con la aplicación. En el repositorio del proyecto [1] se encuentran instrucciones actualizadas para la descarga de la última versión; es posible que estos controles hayan sido alterados:

- ESC: salir de la aplicación.
- ALT: bloquear / liberar el cursor.
- V: bloquear / liberar rotación (y activar bordes naranjas).
- M: activa / desactiva vista inferior.
- N: intercambia vista cenital y primera persona.
- *Comma* y +/-: edita el zoom de la cámara cenital.
- *Period* y +/-: edita el tamaño de la vista inferior.

- 1 hasta 9 y O: activa / desactiva *propiedades* de los portales.
- L y +/-: edita la cantidad máxima de niveles de recursión.
- K y +/-: edita la cantidad de niveles de recursión *saltados*.
- J y +/-: edita el grosor de los portales.

- WASD + movimiento del RATÓN: movimiento libre en primera persona.
 - Rueda del RATÓN: zoom de la cámara principal (click reseteo del zoom).
- TAB: intercambio entre objetos controlables (jugador y portales).
 - TAB y B: intercambio de control entre cámara cenital y jugador.
- FLECHAS: movimiento sobre el plano global XZ del objeto controlado.
- ESPACIO / C: movimiento sobre el plano global Y del objeto controlado.
- R + Q / E: habilitada rotación sobre el objeto controlado y ROLL.
 - Con la rotación activada WASD ejerce PITCH y YAW
- SHIFT / CONTROL: modificadores de la velocidad del movimiento / rotación.
- T: reseteo de posición / rotación del objeto controlado al inicio de su control.
- Y: reseteo de posición / rotación del objeto controlado al inicio de la escena.
- G / H: guarda y carga de transformación completa del objeto controlado.
 - H y B simultáneos: carga la posición guardada de todos los objetos.

- 1 hasta 9: activa / desactiva *elementos* de la escena.
- 1 hasta 9 y P: selecciona el filtro de post-procesado de los portales.
- 1 hasta 9 y B: selecciona el filtro de post-procesado de la escena completa.

- F y +/-: edita el grosor del cuerpo del jugador.
- X / Z: intercambia la cámara principal con las virtuales.
- I: muestra / oculta ejes de coordenadas de jugador y portales.
 - I y O: muestra / oculta ejes de coordenadas de las cámaras virtuales.
 - I y F: muestra / oculta el cuerpo del jugador